Efficient Algorithms for Generating Elliptic Curves over Finite Fields Suitable for Use in Cryptography

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr.rer.nat.)

von

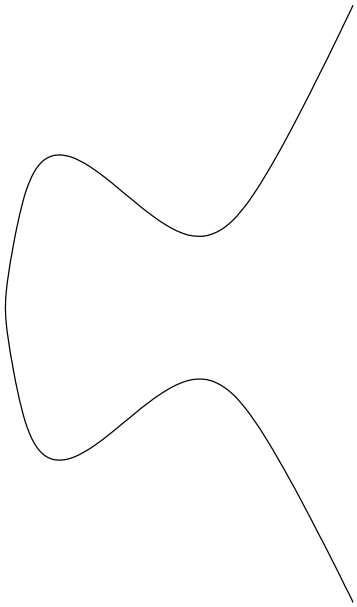
Harald Baier

aus Fulda (Hessen)

Referenten:	Prof. Dr. J. Buchmann
	Prof. Dr. G. Köhler
Tag der Einreichung:	26.03.2002
Tag der mündlichen Prüfung:	07.05.2002

Darmstadt 2002
D 17

To my father.
To my brother.
Grateful for their second lives.



Acknowledgments

Prof. Dr. Johannes Buchmann for being an interested, encouraging, and promoting supervisor. For enabling and supporting my work and this thesis.

Prof. Dr. Günter Köhler for accompanying me through my mathematical education for many years. For accepting the task of the second referee.

My companion Nicole Möller for her love and her patience.

My family for making possible my academic education and her ubiquitous encouragement.

Ulrich Vollmer for reviewing the first part.

Christoph Ludwig for his invaluable advice in C++ and LaTeX.

Thank you all.

Zusammenfassung

Gegenstand der vorliegenden Arbeit ist die Beschreibung eines effizienten Algorithmus zum Auffinden einer elliptischen Kurve über einem endlichen Primkörper $\mathbb{F}_p$, deren Punktegruppe über $\mathbb{F}_p$ kryptographisch geeignet ist.

Der Algorithmus basiert auf der Theorie der Komplexen Multiplikation. Für kryptographische Zwecke sind nur elliptische Kurven relevant, deren Endomorphismenring eine imaginärquadratische Ordnung ist. Wir schreiben $\mathcal{O}_\Delta$ für eine solche Ordnung. Die Theorie der Komplexen Multiplikation erlaubt es zu entscheiden, ob zu vorgegebener Primzahlpotenz q, positiver, ganzer Zahl N und Endomorphismenring $\mathcal{O}_\Delta$ eine elliptische Kurve über $\mathbb{F}_q$ existiert, deren Punktegruppe über $\mathbb{F}_q$ die Ordnung N hat und deren Endomorphismenring gleich $\mathcal{O}_\Delta$ ist. Im Falle der Existenz liefert die Theorie der Komplexen Multiplikation ferner ein Verfahren, um zu gegebenen q, N und $\mathcal{O}_\Delta$ eine solche Kurve zu bestimmen.

Erste Arbeiten zur Erzeugung kryptographisch geeigneter, elliptischer Kurven mittels der Theorie der Komplexen Multiplikation stammen von A.-M. Spallek ([Spa92]) und Lay/Zimmer ([Lay94], [LZ94]). Diese Arbeiten beschreiben jedoch keinen geschlossenen Algorithmus, der bei Eingabe von gewünschten Sicherheitsparametern eine entsprechende Kurve auffindet. Ferner lassen die Arbeiten eine Vielzahl von Fragen ungeklärt, denen wir uns in dieser Arbeit widmen.

Zunächst wird in keiner der Arbeiten beschrieben, wie man effizient einen endlichen Körper $\mathbb{F}_p$ *und* eine Ordnung einer kryptographisch geeigneten Punktegruppe über $\mathbb{F}_p$ bestimmt. Abhängig davon, ob p vorgegeben ist oder nicht, entwickeln wir jeweils effiziente Algorithmen für diese Aufgabe.

Unser Algorithmus berücksichtigt ferner die Klassenzahl der zu dem Endomorphismenring gehörenden maximalen Ordnung. Um nämlich Kurven zu generieren, die den Kriterien des deutschen Signaturgesetzes genügen, muss diese Klassenzahl z.Zt. größer oder gleich 200 sein.

Darüberhinaus entwickeln wir ein effizientes Verfahren zur Berechnung von Klassenpolynomen. Die Berechnung solcher Klassenpolynome stellt einen wichtigen Teilalgorithmus dar, da der Grad des zu berechnenden Klassenpolynoms mitentscheidend für die Gesamtlaufzeit ist. Bisher galten lediglich Polynome vom Grad maximal 50 als verwendbar für Erzeugungsverfahren basierend auf der Theorie der Komplexen Multiplikation (siehe z.B. [MP97]). Wir zeigen, dass wir sogar Klassenpolynome vom Grad 3000 in vertretbarer Zeit berechnen können, also mit einer Laufzeit von weniger als 10 Minuten auf einem handelsüblichen PC.

Ferner untersuchen wir die zur Berechnung eines Klassenpolynoms hinreichende Präzision und geben entsprechende Formeln an. Eine solche praktische Untersuchung ist bisher nicht bekannt. Im Fall eines Klassenpolynoms, das von Yui/Zagier ([YZ97]) vorgeschlagen wurde, entwickeln wir eine neue Formel, die in der Praxis eine Laufzeitverbesserung von bis zu 45% gegenüber bisherigen Vorschlägen ergibt.

Schließlich erweitern wir unseren Algorithmus zur Bestimmung von elliptischen Kurven über Optimalen Erweiterungskörpern, deren Punktegruppe von Primzahlordnung ist. Außerdem entwickeln wir einen Algorithmus, der eine elliptische Kurve über $\mathbb{F}_p$ ausgibt, so dass die Punktegruppen der Kurve und eines Twists über $\mathbb{F}_p$ zyklisch von Primzahlordnung sind.

Alle in dieser Arbeit entwickelten Algorithmen sind in C++ programmiert und in dem LiDIA-Modul gec verfügbar.

Abstract

The subject of the thesis at hand is the description of an efficient algorithm for finding an elliptic curve over a finite prime field of large characteristic suitable for use in cryptography. The algorithm is called `cryptoCurve`. It makes use of the theory of complex multiplication.

Our work relies on proposals of A.-M. Spallek ([Spa92]) and G.J. Lay/H.G. Zimmer ([Lay94], [LZ94]). However, their work leaves several important questions and problems unanswered.

First, neither author presents an algorithm to find a suitable cardinality, that is a prime field *and* a cardinality of a suitable elliptic curve group. We develop and describe a very efficient algorithm for this task; in addition, we give upper bounds of its complexity. In this efficient algorithm the prime field may not be chosen in advance. However, in some cases the field is given first. For instance, all international cryptographic standards which describe an algorithm for finding a suitable cardinality, make use of the latter approach ([P1363], Chapter A.14.2.3, p. 155, [X9.62], Chapter E.3.2.c, p. 115-116). We show how to significantly speed up these algorithms.

Second, no previously proposed algorithm for the generation of an elliptic curve considers the class number of the endomorphism ring of the curve. The German Information Security Agency requires the class number of the maximal order containing the endomorphism ring to be at least 200 ([GIS01]). Our algorithm `cryptoCurve` respects this condition.

Third, we develop and thoroughly investigate different methods to compute class polynomials. The computation of a class polynomial is an important subalgorithm in the complex multiplication approach. In general the integer coefficients of a class polynomial are very large. Hence their computation in practice is rather difficult. It was believed in the cryptographic community that only class polynomials of low degree, say of degree at most 50, are amenable to the complex multiplication approach (see for example [MP97]). However, using our efficient algorithm, we are able to compute a class polynomial of degree up to 3000 in reasonable time, that is in less than 10 minutes on an ordinary PC. In addition, we are able to compute a class polynomial of degree 15000 on the same computer in less than two days.

Fourth, we carry out a detailed practical investigation of the floating point precision needed to compute a class polynomial. The precision in use is important for the run time to compute a class polynomial in practice. However, in order to get a correct result, we have to choose the floating point precision with care. As of today, different precisions were proposed (see for instance [AM93], [BSS99], [LZ94]). All of them are only based on heuristic arguments, and none of the authors presents a practical investigation. In addition, none of the cryptographic standards [P1363] or [X9.62] gives a hint on how to choose an appropriate floating point precision. For instance, we quote from [P1363], Annex A, p. 151: "The above computation must be performed with sufficient accuracy to identify each coefficient of the polynomial $w_D(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than 1/2." Obviously this statement is not useful for choosing the floating point precision in practice. Furthermore, in case of the class polynomial due to N. Yui and D. Zagier ([YZ97]), which uses Weber functions, we propose a new floating point precision to compute this polynomial in practice. Our precision yields a significant performance improvement. Sample tests show an acceleration of about 45% in practice compared to the precision proposed in [LZ94].

All algorithms of this thesis are implemented in C++ and available via the LiDIA module gec.

Contents

List of Algorithms

List of Tables

List of Figures

Part I

Introduction

and

Mathematical Background

Chapter 1

Introduction

1.1 Preface

The subject of the thesis at hand is the description of an efficient algorithm for finding an elliptic curve over a finite prime field of large characteristic suitable for use in cryptography. The algorithm is called `cryptoCurve`.

The group of rational points of an elliptic curve over a field F of characteristic not in $\{2; 3\}$ is the set of solutions of an equation of the form $y^2 = x^3 + ax + b$ over the field F, where both a and b are elements of F. The formal definition of these terms will be given in Section 2.3.

Although elliptic curves were studied thoroughly for a dozen of decades in the framework of number theory and algebraic geometry, their applications in practice evolved rather recently. For instance, in 1987 H.W. Lenstra proposed a new algorithm for factoring large integers using elliptic curves over factor rings of integers ([Len87]). In addition, A.O.L. Atkin and F. Morain, using ideas of S. Goldwasser and J. Kilian ([GK86]), developed and implemented an algorithm for primality proving that employs elliptic curves ([Mor88], [AM93]). Finally, N. Koblitz ([Kob87]) and V.S. Miller ([Mil86]) independently proposed the use of rational points of an elliptic curve defined over a finite field in cryptography.

Once we know that the set of rational points of an elliptic curve over a finite field actually is an Abelian group, we may define the discrete logarithm problem in this group as usual. However, since the use of elliptic curves in cryptography, various algorithms to solve the discrete logarithm problem in the group of rational points of an elliptic curve have been found. Hence, in order to keep the discrete logarithm problem intractable, we have to choose the elliptic curve diligently.

As of today the security of an elliptic curve cryptosystem is determined by the cardinality of the group of rational points of the elliptic curve in use. Thus in order to decide whether a group of rational points is suitable for use in cryptography, we have to know its group order. It turns out that in general this is a burdensome and nontrivial task. If the elliptic curve is defined over a finite prime field of large characteristic, two approaches have been proposed to find a curve with group of rational points which is suitable for use in cryptography:

The first approach, mostly referred to as the *random approach,* first chooses random parameters a and b. Using point counting algorithms, the group order of the set of rational points is determined. Once the cardinality is known, we can decide whether the group is suitable for

use in cryptography or not. If it turns out that the curve does not yield a secure cryptosystem, new random parameters a and b are chosen. The most efficient algorithm to count the number of points of an elliptic curve group over a large finite prime field is due to R. Schoof, N. Elkies, and A.O.L. Atkin, the SEA-algorithm. It has been successfully implemented and improved by V. Müller ([Mül95]). The theoretical complexity of the SEA-algorithm can be shown to be $O(\log^{4+\varepsilon} p)$, where p is the cardinality of the prime field and $\varepsilon > 0$. Although the SEA-algorithm is polynomial in the bitlength of p with low degree, we have to choose various parameters a and b before finding suitable ones, and in either case we have to apply the SEA-algorithm.

However, relying on the work of A.O.L. Atkin and F. Morain, A.-M. Spallek proposed a further approach to find elliptic curve groups for use in cryptography ([Spa92]). This approach uses the theory of complex multiplication. Hence this approach is called the *complex multiplication approach.* Its proceeding is quite different from the random approach. In the complex multiplication method one first searches for candidates of a suitable group cardinality. This can be done without knowing the parameters a and b of the corresponding elliptic curves. Once a suitable cardinality is found, the parameters a and b are determined using complex multiplication. The ideas of Spallek were extended by G.J. Lay and H.G. Zimmer ([Lay94], [LZ94]). However, their work leaves several important questions and problems unanswered.

First, neither author presents an algorithm to find a suitable cardinality, that is a prime field *and* a cardinality of a suitable elliptic curve group. We develop and describe a very efficient algorithm for this task; in addition, we give upper bounds of its complexity. In this efficient algorithm the prime field may not be chosen in advance. However, in some cases the field is given first. For instance, all international cryptographic standards which describe an algorithm for finding a suitable cardinality, make use of the latter approach ([P1363], Chapter A.14.2.3, p. 155, [X9.62], Chapter E.3.2.c, p. 115-116). We show how to significantly speed up these algorithms.

Second, no previously proposed algorithm for the generation of an elliptic curve considers the class number of the endomorphism ring of the curve. The German Information Security Agency requires the class number of the maximal order containing the endomorphism ring to be at least 200 ([GIS01]). Our algorithm `cryptoCurve` respects this condition.

Third, we develop and thoroughly investigate different methods to compute class polynomials. The computation of a class polynomial is an important subalgorithm in the complex multiplication approach. In general the integer coefficients of a class polynomial are very large. Hence their computation in practice is rather difficult. It was believed in the cryptographic community that only class polynomials of low degree, say of degree at most 50, are amenable to the complex multiplication approach (see for example [MP97]). However, using our efficient algorithm, we are able to compute a class polynomial of degree up to 3000 in reasonable time, that is in less than 10 minutes on an ordinary PC. In addition, we are able to compute a class polynomial of degree 15000 on the same computer in less than two days.

Fourth, we carry out a detailed practical investigation of the floating point precision needed to compute a class polynomial. The precision in use is important for the run time to compute a class polynomial in practice. However, in order to get a correct result, we have to choose the floating point precision with care. As of today, different precisions were proposed (see for instance [AM93], [BSS99], [LZ94]). All of them are only based on heuristic arguments, and none of the authors presents a practical investigation. In addition, none of the cryptographic

standards [P1363] or [X9.62] gives a hint on how to choose an appropriate floating point precision. For instance, we quote from [P1363], Annex A, p. 151: "The above computation must be performed with sufficient accuracy to identify each coefficient of the polynomial $w_D(t)$. Since each such coefficient is an integer, this means that the error incurred in calculating each coefficient should be less than 1/2." Obviously this statement is not useful for choosing the floating point precision in practice. Furthermore, in case of the class polynomial due to N. Yui and D. Zagier ([YZ97]), which uses Weber functions, we propose a new floating point precision to compute this polynomial in practice. Our precision yields a significant performance improvement. Sample tests show an acceleration of about 45% in practice compared to the precision proposed in [LZ94].

Finally, in order to be easily usable, we integrate `cryptoCurve` in the library LiDIA ([LiDIA]). As LiDIA is implemented in C++, we make use of an object oriented approach, that is we design a class hierarchy to implement `cryptoCurve`. The corresponding LiDIA-module is called gec - generate elliptic curves. The gec-module provides a broad functionality and an easy-to-use interface. Furthermore, the interface structure of gec is designed in a way that the user can easily adapt gec for his needs. gec may be used as any other class in LiDIA. In addition, we provide an interface to Java; thus once LiDIA is compiled successfully on the platform in use, elliptic curves for use in cryptography may be generated from within the Java Cryptography Architecture.

We tested our algorithms on three platforms: First, a SUN UltraSPARC-I running Solaris 2.6 at 167 MHz and having 256 MByte main memory. Second, a SUN UltraSPARC-IIi running Solaris 2.6 at 333 MHz and having 512 MByte main memory. Finally, a Pentium III running Linux 2.2.14 at 800 MHz and having 128 MByte main memory. All algorithms are implemented in C++ using the library LiDIA 2.0 ([LiDIA]) with gmp 2.0.2 as underlying multiprecision package. We compiled our library and test programs with the GNU compiler 2.95.2 setting the optimization flag O2. Sample tests indicate that running times on the Pentium III are about a quarter of the timings on the SUN UltraSPARC-IIi.

1.2 Structure of the Thesis

The thesis is divided in three parts. The first part comprises this introduction and the mathematical background of our algorithm. This background is presented in Chapter 2. As the theory of complex multiplication is closely related to imaginary quadratic orders and class field theory, we first introduce the relevant theory. Next we turn to the underlying theory of elliptic curves, that is we discuss the relevant properties of elliptic curves over a general field, over the complex numbers, and over a finite field, respectively. In addition, we present an introduction to the theory of complex multiplication. Finally, we list the requirements for a group of rational points of an elliptic curve to be suitable for use in cryptography. We call the cardinality of such a group a *suitable cardinality*.

Next, in part II we describe in detail our generating algorithm. Part II contains the Chapters 3 to 10. First, in Chapter 3 we give an overview of our algorithm. The main algorithm splits into various subalgorithms discussed in the subsequent Chapters.

In Chapter 4 we develop a very efficient algorithm to find a finite prime field and a suitable cardinality. The approach of Chapter 4 is to first choose an imaginary quadratic discriminant. This discriminant is equal to the discriminant of the endomorphism ring of the elliptic curve

which is the output of `cryptoCurve`. A similar idea is due to Spallek ([Spa92]), and a non-optimized variant may be found in [SScK01]. It turns out that the implementation of [SScK01] is much slower than our implementation. As the discriminant is kept fixed in the subalgorithm, we call this approach "Fixed Discriminant Approach". We derive various conditions which have to be respected in order to be successful. These requirements and their efficient implementation yield a highly efficient algorithm to find a suitable cardinality. In addition, we provide the complexity of a non-optimized variant. Hence, we give an upper bound of the complexity of our optimized subalgorithm.

Chapter 5 presents a subalgorithm which determines a suitable cardinality if the prime field is given; we call this variant "Fixed Field Approach". Using ideas of Atkin and Morain ([AM93]), we show how to implement this approach very efficiently in practice.

In Chapter 6 we compare two approaches to compute the reduced representatives of the class group of an imaginary quadratic order. It turns out that the well-known brute-force approach is optimal in practice.

Chapter 7 compares the running times of the computation of various number-theoretical functions in practice. For instance, we compare the running times of different representations to compute the Dedekind η-function within a given floating point precision. In addition, we show that representing the further number-theoretical functions in terms of the η-function is the most efficient way to evaluate them in practice. The main results of this Chapter are already published in [Bai01b] and [Bai01a].

In Chapter 8 we compare different algorithms to compute a class polynomial once its roots are obtained. We generalize an idea due to Karatsuba ([Kar95]) and show that this approach is superior to the standard algorithms.

In Chapter 9 we investigate the floating point precision needed to compute a class polynomial in practice. In case of the ring class polynomial, our practical tests give evidence that a formula due to Lay and Zimmer ([LZ94]) is appropriate. Furthermore, in case of the class polynomial due to Yui and Zagier ([YZ97]), which uses Weber functions, we propose a new formula of the floating point precision. In addition, basing on a large database of practical tests we show that this precision is sufficient to compute this polynomial in practice.

Next, in Chapter 10 we determine an upper bound of the complexity of our generating algorithm. In addition, we discuss some security considerations related to our algorithm. More precisely, we investigate the number of different elliptic curves which may be generated by our algorithm. The term *different* depends on the attacks we take into account. We develop two attacker models and show that in either model the number of different curves is sufficiently large.

Finally, the Chapters 11 and 12 form the closing part III. In Chapter 11 we first provide practical performance results proving the efficiency of our algorithms. For instance, we show that our algorithm finds a prime p and parameters a and b of an elliptic curve in about 12 seconds on an ordinary PC. The group of rational points of this curve over $\mathbb{F}_p$ is of suitable cardinality. Furthermore, we describe two extensions of our algorithm. First, we show how to efficiently find an elliptic curve over a finite prime field such that both the curve and a twist of it are of prime order and suitable for use in cryptography. An application of this algorithm is a pseudo random number generator using elliptic curves (see [Kal86], [Kal88], [Lip00]). Second, we show how to efficiently find an elliptic curve over an Optimal Extension Field suitable for use in cryptography (see also [Bai01c], [Bai01d]).

Chapter 12 describes the class hierarchy and the interface of our library gec. In addition, it presents the interface of gec to Java. Furthermore, we mention two applications of this interface: a Graphical User Interface and a webinterface.

The final appendix gives additional information to some Chapters and presents various sample elliptic curves suitable for use in cryptography. In addition, the appendix contains the manual of our library gec.

1.3 Notation

We outline the notation used in this thesis. Some symbols have different meanings; however, the meaning becomes evident in the respective context.

First, we introduce some notation for integers with special relevance.

p	a rational prime ≥ 5
p_j	the j-th odd prime, that is $p_1 = 3$, $p_2 = 5$, ...
q	a power of p, i.e. $q = p^n$
r	the cryptographic prime factor
k	the cofactor

We next describe symbols used for sets of numbers.

$\mathbb{N}$	the set of positive rational integers
$\mathbb{Z}$	the ring of rational integers
$\mathbb{P}$	the set of positive rational primes
$\mathbb{Q}$	the field of rational numbers
$\mathbb{R}$	the field of real numbers
$\mathbb{C}$	the field of complex numbers
$\widehat{\mathbb{C}}$	the set $\mathbb{C} \cup \{\infty\}$, i.e. the projective closure of $\mathbb{C}$
$\mathfrak{h}$	the upper complex half plane
F	a field
$\overline{F}$	the algebraic closure of a field F
$F^\times$	the multiplicative group of a field F
K	an imaginary quadratic number field
L	a ring class field
$\mathbb{F}_q$	the finite field of q elements
R	an integral domain, i.e. a commutative ring with 1 having no zero divisors

In addition, we introduce the following symbols in the context of lattices, imaginary quadratic orders, and ring of integers.

$\mathcal{L}$	a lattice in $\mathbb{C}$
Δ	an imaginary quadratic discriminant
Δ_K	a fundamental imaginary quadratic discriminant
$\mathcal{O}_\Delta$	the imaginary quadratic order of discriminant Δ
$\mathfrak{a}$	a fractional ideal of an imaginary quadratic order
$I(\mathcal{O}_\Delta)$	the set of invertible fractional ideals of $\mathcal{O}_\Delta$
$P(\mathcal{O}_\Delta)$	the set of principal invertible fractional ideals of $\mathcal{O}_\Delta$
$h(\Delta)$	the class number of discriminant Δ

$C(\Delta)$	the set of all reduced representatives of ideal classes of discriminant Δ
$h_p(\Delta)$	the number of reduced representatives (a, b, c) of discriminant Δ with $b \geq 0$
$\mathcal{O}_F$	the ring of integers in a number field F
$N(\alpha)$	the norm of an element $\alpha \in \mathcal{O}_F$
$\mathcal{N}(\mathfrak{I})$	the norm of an (integral) ideal $\mathfrak{I} \subset \mathcal{O}_F$ (p. 17)

We will make use of the following symbols to represent a class polynomial.

C	a class polynomial
H	a ring class polynomial
G	a class polynomial corresponding to γ_2 due to Atkin and Morain
W	a class polynomial corresponding to Weber functions due to Yui and Zagier

Furthermore, we introduce some notation in the context of elliptic curves.

E	an elliptic curve over a field
(a, b)	the parameters of an elliptic curve
$E(F)$	the group of rational points of E over a field F
O	the point at infinity
$\Delta(E)$	the discriminant of an elliptic curve $E = (a, b)$, that is $\Delta(E) = -16(4a^3 + 27b^2)$
$j(E)$	the j-invariant of an elliptic curve E
$\mathcal{E}$	an elliptic curve over a number field
G	a base point of an elliptic curve cryptosystem

Finally, we define some well-known functions and relations.

$\log x, x \in \mathbb{R}^+$	the natural logarithm
$\lg x, x \in \mathbb{R}^+$	the logarithm to the base 10
$\pi(n), n \in \mathbb{N}$	the number of rational primes $\leq n$
$\mathrm{li}(n), n \in \mathbb{N}$	logarithmic integral: $\mathrm{li}(n) = \int_2^n \frac{dt}{\log t}$
χ_Δ	the quadratic character associated to Δ (p. 14)
$L(s, \chi_\Delta)$	the Dirichlet L-series associated to χ_Δ (p. 14)
$a = b \bmod N$, $a, b \in \mathbb{Z}, N \in \mathbb{N}$	the unique integer a, $0 \leq a \leq N-1$, with $a \equiv b \bmod N$
$f(n) \sim g(n), f, g : \mathbb{N} \to \mathbb{R}^+$	asymptotically equal, that is $\lim_{x\to\infty} \frac{f(n)}{g(n)} = 1$
$\in_R$	choose an element randomly

Chapter 2

Mathematical Background

In this chapter we introduce the underlying mathematics of our generating algorithm. The algorithm is called `cryptoCurve`. The main theory of `cryptoCurve` is the concept of complex multiplication discussed in Section 2.3.3. However, this theory links different areas of number theory and algebra. For instance, complex multiplication is closely related to imaginary quadratic orders and class field theory. Hence in Section 2.1 we first present the theory of imaginary quadratic orders. Next, in Section 2.2 we turn to that part of class field theory which is relevant in our context. Finally, Section 2.3 deals with the theory of elliptic curves related to our algorithm.

2.1 Imaginary Quadratic Orders and Quadratic Forms

One basic object in `cryptoCurve` is an imaginary quadratic order. This section presents the theory of imaginary quadratic orders which is relevant to explain and understand our algorithm `cryptoCurve`. In addition, we introduce quadratic forms which play a crucial role in one of our subalgorithms. Furthermore, we show that quadratic forms and fractional ideals of imaginary quadratic orders are essentially the same number-theoretical object.

This section is organized as follows: First, in Section 2.1.1 we introduce imaginary quadratic orders and present their basic properties which are important in our context. Next, in Section 2.1.2 we discuss quadratic forms and show their close relation to quadratic orders. Finally, Section 2.1.3 works out some results on representing integers by quadratic forms. We will make use of these results to improve the efficiency of `cryptoCurve`.

2.1.1 Imaginary Quadratic Orders

We introduce the basic terms and basic facts in the framework of imaginary quadratic orders. As this theory is well studied, we present all theorems without proof. For more details we refer to [BS66], [Buc01], [Coh95], [Cox89], or [Lan70].

We begin our study with the definition of an imaginary quadratic order. Recall that the ring of integers of a finite extension F of $\mathbb{Q}$ is the set of all elements of F whose minimal polynomial over $\mathbb{Q}$ has integer coefficients.

Definition 2.1.1 *An* imaginary quadratic field *denoted by K is a field of degree 2 over $\mathbb{Q}$ with $K \subset \mathbb{C}$ and $K \not\subset \mathbb{R}$. An* imaginary quadratic order *is a subring of the ring of integers of K which contains 1. We denote an imaginary quadratic order by $\mathcal{O}$. The ring of integers of K is called the* maximal imaginary quadratic order *of K. It is denoted by $\mathcal{O}_K$.*

Imaginary quadratic orders may be described using *imaginary quadratic discriminants*. We first explain this term and then present the link in Theorem 2.1.3.

Definition 2.1.2 *An* imaginary quadratic discriminant *denoted by Δ is a negative integer either congruent 0 or 1 modulo 4. In addition, let f denote the largest integer such that Δ/f^2 is an imaginary quadratic discriminant, too. Then f is called the* conductor *of Δ. Furthermore, Δ/f^2 is called the* field discriminant *of the imaginary quadratic field $\mathbb{Q}(\sqrt{\Delta})$. Finally, if $f = 1$ then Δ is said to be* fundamental. *We denote a fundamental discriminant by Δ_K.*

For instance, the largest imaginary quadratic discriminants are -3, -4, -7, -8, and -11. Obviously, all of them are fundamental. However, -12 is not a fundamental discriminant, as $-12 = -3 \cdot 2^2$. Hence -12 is a discriminant of conductor 2. Its corresponding fundamental discriminant is -3. As quoted in the preceding sentence we will often use the shortcut discriminant to denote an imaginary quadratic discriminant.

We are now able to give the link between imaginary quadratic orders and imaginary quadratic discriminants.

Theorem 2.1.3 *Let $\mathcal{O}$ be an imaginary quadratic order. Then there is a unique imaginary quadratic discriminant Δ with*

$$\mathcal{O} = \mathbb{Z}[(\Delta + \sqrt{\Delta})/2] = \mathbb{Z} + (\Delta + \sqrt{\Delta})/2 \cdot \mathbb{Z} \; . \tag{2.1}$$

If $\mathcal{O}$ is maximal, then Δ is fundamental. In addition, suppose $\mathcal{O}_1$ and $\mathcal{O}_2$ to be imaginary quadratic orders of discriminant Δ_1 and Δ_2, respectively, and let f_1 and f_2 be the corresponding conductors. Then $\mathcal{O}_1 \subset \mathcal{O}_2$ if and only if $\Delta_2 \mid \Delta_1$ and $\Delta_1/f_1^2 = \Delta_2/f_2^2$.

We denote the imaginary quadratic order of discriminant Δ by $\mathcal{O}_\Delta$. We will also make use of the notation $\mathcal{O}$ for the imaginary quadratic order of discriminant Δ if the discriminant of $\mathcal{O}$ is obvious from the context. It is easy to see that $\mathcal{O}_\Delta = \mathbb{Z}[\sqrt{\Delta}]$ or $\mathcal{O}_\Delta = \mathbb{Z}[(1 + \sqrt{\Delta})/2]$ depending on whether $\Delta \equiv 0 \bmod 4$ or $\Delta \equiv 1 \bmod 4$, respectively. If $c_1, c_2 \in \mathbb{C}$, c_1 and c_2 linear independent over $\mathbb{Z}$, we make use of the notation $[c_1, c_2] := c_1\mathbb{Z} + c_2\mathbb{Z}$.

We next turn to ideals of an order $\mathcal{O}_\Delta$. From Definition 2.1.1 it is clear that $\mathcal{O}_\Delta$ is a commutative ring with 1 having no zero divisors, that is $\mathcal{O}_\Delta$ is an *integral domain*. Let $I \subset \mathcal{O}_\Delta$ be an ideal of $\mathcal{O}_\Delta$; we call such an ideal an *integral ideal*. It is well known that every integral ideal of $\mathcal{O}_\Delta$ may be written as

$$I = m\left(a\mathbb{Z} + \frac{b + \sqrt{\Delta}}{2}\mathbb{Z}\right) \; , \tag{2.2}$$

where $a, m \in \mathbb{N}$, $b \in \mathbb{Z}$, $c := (b^2 - \Delta)/(4a) \in \mathbb{Z}$, and $\gcd(a, b, c) = 1$. However, in general we have to deal with *fractional ideals*.

Definition 2.1.4 *Let $\mathcal{O}_\Delta$ be an imaginary quadratic order and K its field of fractions. A* fractional ideal *of $\mathcal{O}_\Delta$ denoted by $\mathfrak{a}$ is a finitely generated $\mathcal{O}_\Delta$-submodule of K. A* principal fractional ideal *is a fractional ideal of the form $\alpha\,\mathcal{O}_\Delta$ for some $\alpha \in K^\times$.*

We remark that integral ideals are obviously fractional. However, in general $\mathfrak{a} \not\subset \mathcal{O}_\Delta$ for a fractional ideal $\mathfrak{a}$ of $\mathcal{O}_\Delta$. Nevertheless, similar to Equation (2.2) for integral ideals, we may write a fractional ideal $\mathfrak{a}$ as

$$\mathfrak{a} = \alpha \left(a\mathbb{Z} + \frac{b+\sqrt{\Delta}}{2}\mathbb{Z} \right) , \tag{2.3}$$

where $\alpha \in K^\times$, $a \in \mathbb{N}$, $b \in \mathbb{Z}$, $c := (b^2-\Delta)/(4a) \in \mathbb{Z}$, and $\gcd(a,b,c) = 1$. The multiplication of elements in K induces a multiplication of fractional ideals of $\mathcal{O}_\Delta$ as usual; if $\mathfrak{a}$ and $\mathfrak{b}$ are fractional ideals of $\mathcal{O}_\Delta$, their product is simply given by $\mathfrak{a}\cdot\mathfrak{b} := \{\sum_{k=1}^{N} a_k b_k : a_k \in \mathfrak{a}, b_k \in \mathfrak{b}, 1 \le k \le N\}$. We next discuss invertible ideals.

Definition 2.1.5 *Let $\mathcal{O}_\Delta$ be an imaginary quadratic order and $\mathfrak{a}$ a fractional ideal of $\mathcal{O}_\Delta$. $\mathfrak{a}$ is* invertible *if there exists a fractional ideal $\mathfrak{b}$ of $\mathcal{O}_\Delta$ with $\mathfrak{a}\cdot\mathfrak{b} = \mathcal{O}_\Delta$. The set of invertible ideals of $\mathcal{O}_\Delta$ is denoted by $I(\mathcal{O}_\Delta)$. Furthermore, the set of principal non-zero fractional ideals is denoted by $P(\mathcal{O}_\Delta)$.*

Obviously, both $P(\mathcal{O}_\Delta)$ and $I(\mathcal{O}_\Delta)$ form an Abelian group with the multiplication of ideals as group operation. In addition, if $\mathfrak{a} \in P(\mathcal{O}_\Delta)$, that is if $\mathfrak{a} = \alpha\,\mathcal{O}_\Delta$ for some $\alpha \in K^\times$, then $\mathfrak{a} \in I(\mathcal{O}_\Delta)$; it is easy to see that $\mathfrak{b} := 1/\alpha \cdot \mathcal{O}_\Delta$ is a principal fractional ideal with $\mathfrak{a}\cdot\mathfrak{b} = \mathcal{O}_\Delta$. Hence, we have shown $P(\mathcal{O}_\Delta) \subset I(\mathcal{O}_\Delta)$. The factor group $I(\mathcal{O}_\Delta)/P(\mathcal{O}_\Delta)$ is rather important.

Definition 2.1.6 *Let $\mathcal{O}_\Delta$ be an imaginary quadratic order. Then the group $I(\mathcal{O}_\Delta)/P(\mathcal{O}_\Delta)$ is called the* ideal class group *of discriminant Δ. We denote the ideal class group by $Cl(\mathcal{O}_\Delta)$.*

The next theorem states that the group $Cl(\mathcal{O}_\Delta)$ is finite.

Definition and Theorem 2.1.7 *Let Δ be an imaginary quadratic discriminant and let $Cl(\mathcal{O}_\Delta)$ be the ideal class group of discriminant Δ. The cardinality of $Cl(\mathcal{O}_\Delta)$ is finite. It is denoted by $h(\mathcal{O}_\Delta)$ and called the* ideal class number.

Before stating the important theorem on representing the elements of $Cl(\mathcal{O}_\Delta)$, we introduce the term of equivalence of fractional ideals.

Definition 2.1.8 *Let $\mathcal{O}_\Delta$ be an imaginary quadratic order and let K denote its field of fractions. In addition, let $\mathfrak{a}$ and $\mathfrak{b}$ be fractional ideals of $\mathcal{O}_\Delta$, respectively. The ideals $\mathfrak{a}$ and $\mathfrak{b}$ are* equivalent, *if there exists an $\alpha \in K^\times$ with $\mathfrak{a} = \alpha\mathfrak{b}$; we denote this by $\mathfrak{a} \sim \mathfrak{b}$.*

Obviously the relation $\sim$ of Definition 2.1.8 induces an equivalence relation on the set of fractional ideals of $\mathcal{O}_\Delta$. As usual, if $\mathfrak{a}$ is a fractional ideal of $\mathcal{O}_\Delta$, we write $[\mathfrak{a}]$ for its equivalence class in $\mathcal{O}_\Delta$. Let $\alpha(a\mathbb{Z}+(b+\sqrt{\Delta})/2\cdot\mathbb{Z})$ be its representation from Equation (2.3). We conclude that $\mathfrak{a}$ is equivalent to an ideal $\mathfrak{b}$ of the form $[1,(b+\sqrt{\Delta})/(2a)]$. Hence, we have $[\mathfrak{a}] = [\mathfrak{b}]$. In addition, it is easy to see that $[\mathfrak{a}] = [\mathfrak{b}]$ if and only if $\mathfrak{a}P(\mathcal{O}_\Delta) = \mathfrak{b}P(\mathcal{O}_\Delta)$. The following theorem shows that for each element of the ideal class group we find a unique fractional ideal $\mathfrak{a}$ representing the element $\mathfrak{a}P(\mathcal{O}_\Delta)$.

Definition and Theorem 2.1.9 *Let $Cl(\mathcal{O}_\Delta)$ be the ideal class group of discriminant Δ. Let $\mathfrak{a}P(\mathcal{O}_\Delta)$ be an element of $Cl(\mathcal{O}_\Delta)$. There exists an ideal $\mathfrak{b} = \mathbb{Z} + (b + \sqrt{\Delta})/(2a) \cdot \mathbb{Z}$ with the following properties:*
1. $\mathfrak{a}P(\mathcal{O}_\Delta) = \mathfrak{b}P(\mathcal{O}_\Delta)$.
2. $a \in \mathbb{N}$, $b \in \mathbb{Z}$, $c = (b^2 - \Delta)/(4a) \in \mathbb{Z}$, $\gcd(a, b, c) = 1$.
3. $b \leq a \leq c$, *and* $b \geq 0$ *if either* $|b| = a$ *or* $a = c$.
The ideal $\mathfrak{b}$ is called the reduced representative *of the class $\mathfrak{b}P(\mathcal{O}_\Delta)$. We denote its class by (a, b, c).*

Finally, let Δ denote an imaginary quadratic discriminant. If $\Delta \equiv 0 \bmod 4$, then $(1, 0, -\Delta/4)$ obviously is the reduced representative of the ideal class $P(\mathcal{O}_\Delta)$. Furthermore, if $\Delta \equiv 1 \bmod 4$, then $(1, 1, (1 - \Delta)/4)$ is the reduced representative of the ideal class $P(\mathcal{O}_\Delta)$. To sum up, if $b_2 \equiv \Delta \bmod 2$, $b_2 \in \{0, 1\}$, then we may write $(1, b_2, (b_2 - \Delta)/4)$ for the reduced representative of the ideal class $P(\mathcal{O}_\Delta)$.

2.1.2 Quadratic Forms

In this section we describe the basic properties of integral binary quadratic forms of negative discriminants. In addition, we discuss the relation of quadratic forms and quadratic orders. Again, we just summarize the main results. For details and proofs we again refer to [BS66], [Buc01], [Coh95], [Cox89], or [Lan70].

Definition 2.1.10 *Let Δ be an imaginary quadratic discriminant. An integral binary quadratic form of discriminant Δ is a triple $(a, b, c) \in \mathbb{Z}^3$ with $\Delta = b^2 - 4ac$. If $\gcd(a, b, c) = 1$ the integral binary quadratic form (a, b, c) is called* primitive.

We abbreviate the term "integral binary quadratic form" by "quadratic form". Let $f = (a, b, c)$ be a quadratic form. We associate with f the bivariate polynomial $f(X, Y) = aX^2 + bXY + cY^2$. Hence, we have the identity

$$f(X, Y) = (X, Y) \begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix}. \tag{2.4}$$

We write $M(f)$ for the matrix corresponding to f. Obviously the map $f \mapsto M(f)$ is injective, and it is easy to recover f from $M(f)$. Using the special linear group $\mathrm{SL}(2, \mathbb{Z})$, that is the set of all 2×2 matrices of determinant 1 with integer coefficients, we may define a relation on the set of quadratic forms of discriminant Δ as follows.

Definition 2.1.11 *Let f and g be quadratic forms of discriminant Δ. The forms f and g are said to be* equivalent*, if there is a matrix $T \in \mathrm{SL}(2, \mathbb{Z})$ with $M(g) = T^{-1}M(f)T$. If f and g are equivalent, we write $f \sim g$.*

The relation of Definition 2.1.11 clearly defines an equivalence relation on the set of quadratic forms of discriminant Δ. As usual, we denote the equivalence class of f by $[f]$. Similar to Theorem 2.1.9, there is a unique representative for either equivalence class.

Definition and Theorem 2.1.12 *Let Δ be an imaginary quadratic discriminant, and let f be a primitive quadratic form of discriminant Δ. There exists a matrix $T \in \mathrm{SL}(2,\mathbb{Z})$ such that the quadratic form (a,b,c) corresponding to the matrix $T^{-1}M(f)T$ has the following properties:*
1. $f \sim (a,b,c)$.
2. $a \in \mathbb{N}$, $b \in \mathbb{Z}$, $c = (b^2-\Delta)/(4a) \in \mathbb{Z}$, $\gcd(a,b,c) = 1$.
3. $b \le a \le c$, *and* $b \ge 0$ *if either* $|b| = a$ *or* $a = c$.
The quadratic form (a,b,c) is called the reduced representative *of the equivalence class $[f]$.*

For instance, as discussed at the end of Section 2.1.1, if $b_2 \equiv \Delta \bmod 2$, $b_2 \in \{0,1\}$, then $(1, b_2, (b_2-\Delta)/4)$ is a reduced representative. The corresponding quadratic form is called the *principal form* or *main form of discriminant* Δ.

Definition 2.1.13 *Let Δ be an imaginary quadratic discriminant. The set of equivalence classes of primitive quadratic forms of discriminant Δ is called the* form class group *of discriminant Δ. We denote the form class group of discriminant Δ by $Cl(\Delta)$.*

We have to explain that $Cl(\Delta)$ actually is a group. However, in order to explain the group law on equivalence classes of primitive quadratic forms of discriminant Δ, we have to map quadratic forms to fractional ideals of quadratic orders. More precisely, the following theorem states that the form class group $Cl(\Delta)$ and the ideal class group $Cl(\mathcal{O}_\Delta)$ are two sides of the same coin.

Theorem 2.1.14 *Let Δ be an imaginary quadratic discriminant, and let (a,b,c) be a primitive quadratic form of discriminant Δ. Define the map Φ by*

$$\Phi : (a,b,c) \mapsto [a, (-b+\sqrt{\Delta})/2] \,, \tag{2.5}$$

where $\mathfrak{a} := [a, (-b+\sqrt{\Delta})/2]$ is a fractional ideal of $\mathcal{O}_\Delta$. Then $\mathfrak{a}$ is invertible. In addition, Φ induces a bijection between the form class group $Cl(\Delta)$ and the ideal class group $Cl(\mathcal{O}_\Delta)$.

Using the bijection Φ, we get an obvious composition law on $Cl(\Delta)$. Hence, Φ actually induces an isomorphism between $Cl(\Delta)$ and $Cl(\mathcal{O}_\Delta)$. We will refer to both $Cl(\Delta)$ and $Cl(\mathcal{O}_\Delta)$ as the *class group of discriminant* Δ, and we write $Cl(\Delta)$ for the class group. In addition, Theorem 2.1.14 shows $|Cl(\Delta)| = |Cl(\mathcal{O}_\Delta)| = h(\mathcal{O}_\Delta)$. Thus, we set $h(\Delta) := h(\mathcal{O}_\Delta)$ and refer to $h(\Delta)$ as the *class number*. In our algorithm we are only concerned with reduced representatives of form classes. Thus we introduce the following notation.

Definition 2.1.15 *Let Δ be an imaginary quadratic discriminant, and let $Cl(\Delta)$ be the class group. We write $C(\Delta)$ for the set of all reduced representatives of form classes of discriminant Δ.*

We will often refer to the set $C(\Delta)$ as the class group, too. We conclude this section with a deep result concerning the growth of the class number of imaginary quadratic discriminants. This result is a consequence of the analytic class number formula given in Theorem 2.1.17. In order to state the analytic class number formula we have to introduce two terms ([Zag81], p. 38, 41).

Definition 2.1.16 *Let Δ be an imaginary quadratic discriminant.*
1. *The* quadratic character $\chi_\Delta : \mathbb{N} \to \{0; 1; -1\}$ *associated to Δ is defined by*

$$\begin{aligned} \chi_\Delta(p) &\equiv \Delta^{(p-1)/2} \bmod p, \chi_\Delta(p) \in \{0; 1; -1\} && \text{if } p \text{ is an odd prime,} \\ \chi_\Delta(2) &= 0, 1, -1 && \text{for } \Delta \equiv 0 \bmod 4,\ \Delta \equiv 1 \bmod 8, \\ & && \Delta \equiv 5 \bmod 8, \text{ respectively,} \\ \chi_\Delta(nm) &= \chi_\Delta(n)\chi_\Delta(m) && \text{for all } n, m \in \mathbb{N}. \end{aligned}$$

2. *The* Dirichlet L-series *associated to χ_Δ is defined by*

$$L(s, \chi_\Delta) = \sum_{k=1}^{\infty} \frac{\chi_\Delta(k)}{k^s}, \quad s \in \mathbb{C}, \mathrm{Re}(s) > 0.$$

We are now able to state the analytic class number formula ([Zag81], Equation (13) and (14), p. 103).

Theorem 2.1.17 *Let Δ be an imaginary quadratic discriminant. Then*

$$L(1, \chi_\Delta) = \frac{2\pi}{u} \cdot \frac{h(\Delta)}{\sqrt{|\Delta|}} \tag{2.6}$$

where u denotes the number of units in $\mathcal{O}_\Delta$.

It is well known that $L(1, \chi_\Delta)$ is bounded by $\log |\Delta|$. Thus using the O-notation we sum up Theorem 2.1.17 as $|\Delta| = O(h(\Delta)^2)$ and $h(\Delta)^2 = O(|\Delta|)$.

2.1.3 Representing Integers by Quadratic Forms

In our generating algorithm `cryptoCurve` we are concerned with the fundamental problem in number theory to decide whether a given integer n may be represented by a quadratic form or an element of a quadratic order. In this section we prove some results in this context to improve the efficiency of `cryptoCurve`. The results of this section will be used in Chapter 4 to efficiently find primes represented by a given quadratic form. However, we have to explain the term representation

Definition 2.1.18 *Let Δ be an imaginary quadratic discriminant. In addition, let $n \in \mathbb{Z}$ and $f = (a, b, c)$ be a primitive quadratic form of discriminant Δ. n is* represented by f, *if there is a pair $(x, w) \in \mathbb{Z}^2$ with $n = ax^2 + bxw + cw^2 =: f(x, w)$. Furthermore, n is the* norm of an element of $\mathcal{O}_\Delta$, *if there exists an element $\pi \in \mathcal{O}_\Delta$ with $n = \pi\overline{\pi}$.*

From Sections 2.1.1 and 2.1.2 we are already aware of the close relationship between quadratic forms and quadratic orders. In this section we present some basic facts concerning representations of integers by norms of elements of quadratic orders and by quadratic forms. The outlined facts will be very useful in Chapter 4, when we have to find primes for our generating algorithm. We first develop a requirement to check whether a given integer is norm of an element of $\mathcal{O}_\Delta$. It turns out that this property is much easier to check in practice.

Proposition 2.1.19 *Let $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$ be an imaginary quadratic discriminant. Furthermore, let n be an integer. There exists an element $\pi \in \mathcal{O}_\Delta$ with $n = \pi\overline{\pi}$ if and only if there are integers t and y with $4n = t^2 - \Delta y^2$.*

Proof: I. Let $\pi \in \mathcal{O}_\Delta$ satisfy $n = \pi\overline{\pi}$. We have $\mathcal{O}_\Delta = [1, (\Delta+\sqrt{\Delta})/2]$. Thus, there are integers a and b with $\pi = a+b\frac{\Delta+\sqrt{\Delta}}{2}$. It follows $n = \pi\overline{\pi} = (a+b\frac{\Delta+\sqrt{\Delta}}{2})(a+b\frac{\Delta-\sqrt{\Delta}}{2}) = (a+\frac{b\Delta}{2})^2 - b^2\frac{\Delta}{4}$, i.e. $4n = (2a+b\Delta)^2 - b^2\Delta$. If we set $t := 2a+b\Delta$ and $y := b$, we have $4n = t^2 - \Delta y^2$.
II. Now let $4n = t^2 - \Delta y^2$ with integers t and y. It is easy to see that $4n = t^2 - \Delta y^2$ implies $t \equiv y\Delta \bmod 2$, and $\pi := \frac{t-y\Delta}{2} + y\frac{\Delta+\sqrt{\Delta}}{2}$ is an element in $\mathcal{O}_\Delta$ of norm n. □

Next we show that there is no difference in saying "p is the norm of an element of $\mathcal{O}_\Delta$" and "p is represented by the main form of discriminant Δ". However, we first have to introduce some new notation.

Definition 2.1.20 *Let $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$. Furthermore, let f be the main form of discriminant Δ. For an integer n we denote by $N_\Delta(n)$ the set of pairs of integers (t, y) for which we have $4n = t^2 - \Delta y^2$, and we write $R_{\Delta,f}(n)$ for the set of pairs of integers representing n by f,i.e.*

$$N_\Delta(n) = \{(t, y) \in \mathbb{Z}^2 : 4n = t^2 - \Delta y^2\}, \tag{2.7}$$
$$R_{\Delta,f}(n) = \{(x, w) \in \mathbb{Z}^2 : n = f(x, w)\}. \tag{2.8}$$

Proposition 2.1.21 *Let Δ, f, and n be as in Definition 2.1.20. The map ρ defined by*

$$\rho : N_\Delta(n) \to R_{\Delta,f}(n), \quad (t, y) \mapsto \begin{cases} (\frac{t}{2}, y) & : \Delta \equiv 0 \bmod 4 \\ (\frac{t-y}{2}, y) & : \Delta \equiv 1 \bmod 4 \end{cases}$$

is a bijection from $N_\Delta(n)$ to $R_{\Delta,f}(n)$ with inverse

$$\rho^{-1} : R_{\Delta,f}(n) \to N_\Delta(n), \quad (x, w) \mapsto \begin{cases} (2x, w) & : \Delta \equiv 0 \bmod 4 \\ (2x+w, w) & : \Delta \equiv 1 \bmod 4\,. \end{cases}$$

Proof: We have $f = (1, 0, \frac{-\Delta}{4})$ in case of $\Delta \equiv 0 \bmod 4$ and $f = (1, 1, \frac{1-\Delta}{4})$ in case of $\Delta \equiv 1 \bmod 4$. According to Proposition 2.1.19, an integer n is the norm of an element of $\mathcal{O}_\Delta$ if and only if there are integers t and y with $4n = t^2 - \Delta y^2$.
I. We first show $\rho(N_\Delta(n)) \subseteq R_{\Delta,f}(n)$. Let $(t, y) \in N_\Delta(n)$. If $\Delta \equiv 0 \bmod 4$ we have $n = \frac{t^2}{4} - \frac{\Delta}{4}y^2 \in \mathbb{Z}$, i.e. $\frac{t^2}{4} \in \mathbb{Z}$. We set $x = \frac{t}{2}$ and $w = y$; then $n = f(x, w)$ follows. If $\Delta \equiv 1 \bmod 4$, we have $\Delta \equiv 1 \bmod 2$; in addition, $t \equiv y \bmod 2$ holds (see the proof of Proposition 2.1.19). We set $x = \frac{t-y}{2}$ and $w = y$; again $n = f(x, w)$ follows.
II. We prove that the image of the map

$$\lambda : R_{\Delta,f}(n) \to \mathbb{Z}^2, \quad (x, w) \mapsto \begin{cases} (2x, w) & : \Delta \equiv 0 \bmod 4 \\ (2x+w, w) & : \Delta \equiv 1 \bmod 4 \end{cases}$$

is a subset of $N_\Delta(n)$ Let $(x, w) \in R_{\Delta,f}(n)$. We first assume $\Delta \equiv 0 \bmod 4$. It follows $n = x^2 - \frac{\Delta}{4}w^2$, i.e. $4n = (2x)^2 - \Delta w^2$. Therefore, $(2x, w)$ is an integer solution of $4n = t^2 - \Delta y^2$. Next we assume $\Delta \equiv 1 \bmod 4$. Thus $n = x^2 + xw + \frac{1-\Delta}{4}w^2$ and $4n = (2x)^2 + 2\cdot(2x)w + w^2 - \Delta w^2 =$

$(2x+w)^2 - \Delta w^2$ follow. Hence, $(2x+w, w)$ is an integer solution of $4n = t^2 - \Delta y^2$.
III. Finally we prove $\lambda \circ \rho = \mathrm{id}_{N_\Delta(n)}$ and $\rho \circ \lambda = \mathrm{id}_{R_{\Delta,f}(n)}$. From these identities we deduce that ρ and $\lambda = \rho^{-1}$ are bijective.
For a pair $(t, y) \in N_\Delta(n)$ we have

$$\lambda \circ \rho(t,y) = \begin{cases} \lambda(\frac{t}{2}, y) = (2 \cdot \frac{t}{2}, y) = (t,y) & : \ \Delta \equiv 0 \bmod 4 \\ \lambda(\frac{t-y}{2}, y) = (2 \cdot \frac{t-y}{2} + y, y) = (t,y) & : \ \Delta \equiv 1 \bmod 4. \end{cases}$$

This proves $\lambda \circ \rho = \mathrm{id}_{N_\Delta(n)}$. For $(x, w) \in R_{\Delta,f}(n)$ we have

$$\rho \circ \lambda(x,w) \begin{cases} \rho(2x, w) = (\frac{2x}{2}, w) = (x,w) & : \ \Delta \equiv 0 \bmod 4 \\ \rho(2x+w, w) = (\frac{2x+w-w}{2}, w) = (x,w) & : \ \Delta \equiv 1 \bmod 4 \end{cases}$$

and $\rho \circ \lambda = \mathrm{id}_{R_{\Delta,f}(n)}$ follows. □

Hence, Proposition 2.1.19 and Proposition 2.1.21 imply that a rational prime p is the norm of an element of $\mathcal{O}_\Delta$ if and only if p is represented by the main form of discriminant Δ. Finally, we show that there is a one-to-one correspondence between positive integers t and y and primes of the form $(t^2 - \Delta y^2)/4$. We will need this theorem in Section 4.2 to determine the complexity of one of our subalgorithms to find a suitable prime.

Theorem 2.1.22 *Let Δ be an imaginary quadratic discriminant with fundamental discriminant $\Delta_K < -4$. In addition, let t and y be positive integers such that $p := (t^2 - \Delta y^2)/4$ is an odd prime. Then t and y are uniquely determined by p.*

Proof: Let $(t, y) \in \mathbb{N}^2$ with $p = (t^2 - \Delta y^2)/4$. The proof of Proposition 2.1.19 shows that each of the four pairs $(\pm t, \pm y)$ yields a different element of $\mathcal{O}_\Delta$ of norm p. Thus there are at least four elements in $\mathcal{O}_\Delta$ of norm p.
We next prove that there are exactly 4 elements in $\mathcal{O}_\Delta$ of norm p. Let $\pi \in \mathcal{O}_\Delta$ with $p = \pi\overline{\pi}$. Furthermore, let $\pi' \in \mathcal{O}_\Delta$ be a further element of norm p. Hence, we have $p = \pi\overline{\pi} = \pi'\overline{\pi'}$. The last equation shows $(\pi/\pi')\overline{(\pi/\pi')} = 1$. Thus, π/π' is a complex number of norm 1. Let $K = \mathbb{Q}(\sqrt{\Delta})$ denote the field of fractions of $\mathcal{O}_\Delta$. Then we have $\pi/\pi' \in K$. Thus π/π' is a root of unity and hence an element of $\mathcal{O}_{\Delta_K}$. However, the assumption $\Delta_K < -4$ yields $\mathcal{O}_{\Delta_K}^\times = \{1, -1\}$. Hence, $\pi' \in \{\pi, -\pi\}$ follows. Thus there are four elements in $\mathcal{O}_\Delta$ of norm p : π, $-\pi$, $\overline{\pi}$, and $-\overline{\pi}$. □

2.2 Class Field Theory

In this section we introduce the relevant theory of class fields. It turns out that imaginary quadratic orders and this theory are closely linked. The link is given by the theory of complex multiplication.

In Section 2.2.1 we begin our studies with the basic properties of number fields. Using these results we are able to introduce in Section 2.2.2 the most important class fields in our context, *ring class fields*. In one of our subalgorithms we have to compute a polynomial which generates the ring class field. Thus in Section 2.2.3 we discuss three different polynomials suitable for this purpose.

2.2.1 Number Fields

This section summarizes the basic theory of number fields. Similar to the factorization of rational integers into a product of rational primes, the focus of our study lies on the decomposition of an ideal in the ring of integers of a number field into prime ideals. We first turn to the definition of number fields.

Definition 2.2.1 *A* number field *denoted by F is a subfield of the complex numbers $\mathbb{C}$ having finite degree over the rationals $\mathbb{Q}$. The* degree *of F over $\mathbb{Q}$ is denoted by $[F : \mathbb{Q}]$.*

For instance, an imaginary quadratic field K is a number field with $[K : \mathbb{Q}] = 2$. As usual we denote by $\mathcal{O}_F$ the ring of integers of the number field F. In addition to integral ideals of imaginary quadratic orders, we present some properties of $\mathcal{O}_F$-ideals.

Definition and Theorem 2.2.2 *Let F be a number field and $\mathfrak{I} \subset \mathcal{O}_F$ a non-zero $\mathcal{O}_F$-ideal. Then the quotient ring $\mathcal{O}_F/\mathfrak{I}$ is finite. Its cardinality is called the* norm *of $\mathfrak{I}$. The norm of $\mathfrak{I}$ is denoted by $\mathcal{N}(\mathfrak{I})$.*

We remark that if $\mathfrak{I} = \alpha\,\mathcal{O}_F$ is principal, then $\mathcal{N}(\mathfrak{I})$ is equal to the norm $|N_{F/\mathbb{Q}}(\alpha)|$ of the algebraic integer α. We next turn to the unique decomposition of non-zero $\mathcal{O}_F$-ideals into prime ideals. Similar to the main theorem of number theory stating the unique factorization of integers, the following holds in a ring of integers (see [Cox89], Corollary 5.6, p.99).

Theorem 2.2.3 *Let F be a number field and $\mathfrak{I} \subset \mathcal{O}_F$ a non-zero $\mathcal{O}_F$-ideal. Then $\mathfrak{I}$ can be written as a product*

$$\mathfrak{I} = \prod_{k=1}^{N} \mathfrak{p}_k^{e_k}\ , \tag{2.9}$$

where the $\mathfrak{p}_k$ are pairwise different non-zero prime ideals in $\mathcal{O}_F$. The decomposition of Equation (2.9) is unique up to order of the prime ideals $\mathfrak{p}_k$. In addition, the ideals $\mathfrak{p}_k$ are exactly the prime ideals in $\mathcal{O}_F$ containing $\mathfrak{I}$.

According to Theorem 2.2.3 it is common to call a prime ideal $\mathfrak{p} \neq (0)$ a *prime* of F. The primes of number fields play a rather important role in class field theory and complex multiplication. We present a further correspondence between rational primes and primes of a number field. If p is a rational prime, the factor ring $\mathbb{Z}/p\mathbb{Z}$ is the field of p elements. A similar general theorem holds for primes of number fields.

Theorem 2.2.4 *Let F be a number field and $\mathfrak{p}$ a prime of F, that is $\mathfrak{p}$ is a non-zero prime ideal in $\mathcal{O}_F$. Then $\mathcal{O}_F/\mathfrak{p}$ is a finite field.*

However, in contrast to $\mathbb{Z}$, the field $\mathcal{O}_F/\mathfrak{p}$ is in general not a prime field. We next turn to the decomposition of primes of F in finite field extensions E/F. If $\mathfrak{p}$ is a prime of F, then it is common to denote by $\mathfrak{p}\,\mathcal{O}_E$ the ideal generated by $\mathfrak{p}$ in $\mathcal{O}_E$. According to Theorem 2.2.3 we have a unique decomposition

$$\mathfrak{p}\,\mathcal{O}_E = \prod_{k=1}^{N} \mathfrak{P}_k^{e_k}\ , \tag{2.10}$$

where the $\mathfrak{P}_k$ are pairwise different prime ideals in $\mathcal{O}_E$ containing $\mathfrak{p}\,\mathcal{O}_E$. In addition, it is easy to see that $\mathcal{O}_E/\mathfrak{P}_k : \mathcal{O}_F/\mathfrak{p}$ is a finite extension of finite fields. We introduce the following terms.

Definition 2.2.5 *Let E/F be a finite extension of number fields E and F. In addition, let $\mathfrak{p}$ be a prime of F, and $\mathfrak{p}\,\mathcal{O}_E = \prod_{k=1}^{N} \mathfrak{P}_k^{e_k}$ be its factorization in $\mathcal{O}_E$. The integer e_k is called the* ramification index *of $\mathfrak{p}$ in $\mathfrak{P}_k$. Furthermore, if we set $f_k := [\mathcal{O}_E/\mathfrak{P}_k : \mathcal{O}_F/\mathfrak{p}]$, then f_k is named the* inertial degree *of $\mathfrak{p}$ in $\mathfrak{P}_k$.*

In our application, we only have to deal with finite Galois extensions E/F. In this case, there is a close relationship between the degree $[E : F]$, the ramification indices e_k, and the inertial degrees f_k.

Theorem 2.2.6 *Let E/F be a finite Galois extension of number fields E and F. In addition, let $\mathfrak{p}$ be a prime of F, and let $\mathfrak{p}\,\mathcal{O}_E = \prod_{k=1}^{N} \mathfrak{P}_k^{e_k}$ be its factorization in $\mathcal{O}_E$. Then all ramification indices e_k are equal. In addition, all inertial degrees f_k are the same. Let e and f denote the ramification index and inertial degree of $\mathfrak{p}$ in either prime $\mathfrak{P}_k$, respectively. Then*

$$efN \quad = \quad [E : F]\,.$$

Let E, F and $\mathfrak{p}$ be as in Theorem 2.2.6. We say that $\mathfrak{p}$ *ramifies* in E if $e > 1$, and we say that $\mathfrak{p}$ *splits completely* in E, if $f = e = 1$. It is well known that for a given field extension E/F the number of primes of F which ramify in E is finite.

2.2.2 The Ring Class Field

In this section we study ring class fields and a special case of ring class fields, the *Hilbert class fields*. However, in order to define and explore the basic facts of ring class fields, we first present the main results of class field theory. We follow the approach of [Cox89], §8 A. As our algorithm restricts to ring class fields of imaginary quadratic orders, we discuss class field theory under this aspect.

Definition 2.2.7 *Let K be an imaginary quadratic field. A* modulus *of K denoted by $\mathfrak{m}$ is a non-zero ideal in $\mathcal{O}_K$. Furthermore, an ideal $\mathfrak{a} \subset \mathcal{O}_K$ is* relatively prime *to $\mathfrak{m}$, if $\mathfrak{m} + \mathfrak{a} = \mathcal{O}_K$.*

If the modulus $\mathfrak{m}$ is of the form $f\,\mathcal{O}_K$ for some $f \in \mathbb{N}$, then an ideal $\mathfrak{a} \subset \mathcal{O}_K$ is relatively prime to $\mathfrak{m}$ if and only if $\gcd(f, \mathcal{N}(\mathfrak{a})) = 1$. The concept of a modulus yields a generalization of ideal class groups as introduced in Section 2.1.1. Similar to invertible ideals of an order $\mathcal{O}_\Delta$, we introduce sets of fractional ideals of maximal orders as follows.

Definition 2.2.8 *Let K be an imaginary quadratic field, and let $\mathfrak{m}$ be a modulus of K.*
1. *By $I_K(\mathfrak{m})$ we denote the group of all fractional ideals of $\mathcal{O}_K$ generated by ideals $\mathfrak{a} \subset \mathcal{O}_K$ relatively prime to $\mathfrak{m}$.*
2. *In addition, we denote by $P_{K,1}(\mathfrak{m})$ the group of all principal fractional ideals generated by principal ideals $\alpha\,\mathcal{O}_K$ with $\alpha \in \mathcal{O}_K$, $\alpha \equiv 1 \bmod \mathfrak{m}$.*
3. *Finally, let $f \in \mathbb{N}$. We denote by $P_{K,\mathbb{Z}}(f\,\mathcal{O}_K)$ the group of all principal fractional ideals generated by principal ideals $\alpha\,\mathcal{O}_K$ with $\alpha \in \mathcal{O}_K$, $\alpha \equiv a \bmod f\,\mathcal{O}_K$, $\gcd(a, f) = 1$.*

Obviously we have $P_{K,1}(\mathfrak{m}) \subset I_K(\mathfrak{m})$ and $P_{K,1}(f\,\mathcal{O}_K) \subset P_{K,\mathbb{Z}}(f\,\mathcal{O}_K) \subset I_K(f\,\mathcal{O}_K)$ for all $f \in \mathbb{N}$. It is easy to see that $I_K(\mathfrak{m})/P_{K,1}(\mathfrak{m})$ is finite. A subgroup G of $I_K(\mathfrak{m})$ with $P_{K,1}(\mathfrak{m}) \subset G \subset I_K(\mathfrak{m})$ is called a *congruence subgroup* for the modulus $\mathfrak{m}$. In addition, the factor group $I_K(\mathfrak{m})/G$ is called a *generalized ideal class group* for the modulus $\mathfrak{m}$. For instance, let Δ_K be a fundamental imaginary quadratic discriminant, and let $K = \mathbb{Q}(\sqrt{\Delta_K})$. If $\mathfrak{m} = \mathcal{O}_K$, obviously $I_K(\mathfrak{m}) = I(\mathcal{O}_{\Delta_K})$ and $P_{K,1} = P(\mathcal{O}_{\Delta_K})$. Hence the ideal class group $Cl(\mathcal{O}_{\Delta_K})$ is a generalized ideal class group. However, the same holds if Δ is not fundamental, as we will now state.

Theorem 2.2.9 *Let Δ be an imaginary quadratic discriminant of conductor f. Furthermore, let $K = \mathbb{Q}(\sqrt{\Delta})$. Then we have $Cl(\mathcal{O}_\Delta) \simeq I_K(f\,\mathcal{O}_K)/P_{K,\mathbb{Z}}(f\,\mathcal{O}_K)$.*

Theorem 2.2.9 states that we may express the ideal class group of an order by sets of fractional ideals of the corresponding maximal order.

As summarized in [Cox89], "the basic idea of class field theory is that the generalized ideal class groups are the Galois groups of all Abelian extensions of K". A field extension E/K is called Abelian, if it is a Galois extension and if the Galois group is commutative. However, we have to explain how to link generalized ideal class groups to field extensions. The crucial observation is that there is an outstanding K-automorphism of E, the *Artin symbol.* However, we first have to state the underlying theorem.

Definition and Theorem 2.2.10 *Let E/F be a finite Abelian extension of number fields, and let $\mathfrak{p}$ be a prime of F which is unramified in E. If $\mathfrak{P}$ is a prime of E containing $\mathfrak{p}$, then there is a unique element $\sigma \in \mathrm{Gal}(E/F)$ such that for all $\alpha \in \mathcal{O}_E$ we have*

$$\sigma(\alpha) \equiv \alpha^{\mathcal{N}(\mathfrak{p})} \bmod \mathfrak{P}.$$

The automorphism σ only depends on $\mathfrak{p}$ and is independent of the choice of $\mathfrak{P}$. It is called the Artin symbol *and denoted by $\left(\frac{E/F}{\mathfrak{p}}\right)$.*

Let K be an imaginary quadratic field and let E be an Abelian extension of K. In addition, let $\mathfrak{m}$ be a modulus of K which is divisible by all primes of K which ramify in E. Hence, if $\mathfrak{p}$ is a prime not dividing $\mathfrak{m}$, then we may define the Artin symbol $\left(\frac{E/K}{\mathfrak{p}}\right)$. In order to define the Artin map for E/K and $\mathfrak{m}$, we have to extend the Artin symbol to fractional ideals in $I_K(\mathfrak{m})$. Hence, let $\mathfrak{a} \in I_K(\mathfrak{m})$. We may write $\mathfrak{a} = I_1/I_2$, where both I_1 and I_2 are integral ideals in $\mathcal{O}_K$ relatively prime to $\mathfrak{m}$. Thus, from Theorem 2.2.3 we deduce that we may write

$$\mathfrak{a} \;=\; \prod_{k=1}^{N} \mathfrak{p}_k^{e_k} \tag{2.11}$$

with primes $\mathfrak{p}_k$ of K relatively prime to $\mathfrak{m}$ and $e_k \in \mathbb{Z}$. The Artin symbol of $\mathfrak{a}$ is defined as

$$\left(\frac{E/K}{\mathfrak{a}}\right) \;=\; \prod_{k=1}^{N} \left(\frac{E/K}{\mathfrak{p}_k}\right)^{e_k}. \tag{2.12}$$

The Artin map for E/K and $\mathfrak{m}$ denoted by $\Phi_\mathfrak{m}$ maps the fractional ideal $\mathfrak{a}$ to the automorphism of Equation (2.12). It is well known that $\Phi_\mathfrak{m} : I_K(\mathfrak{m}) \to \mathrm{Gal}(E/K)$ is a surjective homomorphism.

The following theorem of class field theory referred to as the Existence Theorem is crucial for defining the ring class field.

Theorem 2.2.11 *Let K be an imaginary quadratic field and $\mathfrak{m}$ be a modulus of K. In addition, let G be a congruence subgroup for $\mathfrak{m}$. Then there is a unique Abelian extension E of K, all of whose ramified primes divide $\mathfrak{m}$, such that if $\Phi_{\mathfrak{m}} : I_K(\mathfrak{m}) \to \mathrm{Gal}(E/K)$ is the Artin map for E/K and $\mathfrak{m}$, then $G = \ker(\Phi_{\mathfrak{m}})$.*

Finally, we are able to define the ring class field and Hilbert class field.

Definition 2.2.12 *Let Δ be an imaginary quadratic discriminant of conductor f. In addition, let $K = \mathbb{Q}(\sqrt{\Delta})$, and let G denote the congruence subgroup $P_{K,\mathbb{Z}}(f\,\mathcal{O}_K)$. According to Theorem 2.2.11 there is a unique Abelian extension L/K which corresponds to G. It is called the* ring class field *of the order $\mathcal{O}_\Delta$. If $f = 1$, L is called the* Hilbert class field.

We conclude this section by discussing some properties of the ring class field. Let Δ be an imaginary quadratic discriminant of conductor f. First, if $\mathfrak{p}$ is a prime of $K = \mathbb{Q}(\sqrt{\Delta})$ which ramifies in the ring class field L of $\mathcal{O}_\Delta$, then $\mathfrak{p}$ divides the modulus $f\,\mathcal{O}_K$. Equivalent to this statement is that if $\mathfrak{p}$ is a prime not dividing $f\,\mathcal{O}_K$, then $\mathfrak{p}$ is unramified in L. Second, Theorem 2.2.9, Theorem 2.2.11, and the remark following Equation (2.12) yield the isomorphism

$$Cl(\mathcal{O}_\Delta) \simeq I_K(f\,\mathcal{O}_K)/P_{K,\mathbb{Z}}(f\,\mathcal{O}_K) \simeq \mathrm{Gal}(L/K)\,. \tag{2.13}$$

Thus we conclude $[L : K] = h(\Delta)$. Finally, it is obvious that we may characterize the Hilbert class field as the maximal unramified extension of the base field K.

2.2.3 Polynomials Generating the Ring Class Field

In this section we present three polynomials in $\mathbb{Z}[X]$ which generate the ring class field L/K. The order of magnitude of their coefficients is crucial for the running time of our generating algorithm `cryptoCurve` in practice. More precisely, in order to speed up `cryptoCurve`, the coefficients should be as small as possible (with respect to their absolute value).

Definition 2.2.13 *Let L/K be the ring class field of an imaginary quadratic order $\mathcal{O}_\Delta$. A monic polynomial $C \in \mathbb{Z}[X]$ is called a* class polynomial *if $K(\alpha) = L$ for any root α of C. A root of a class polynomial C is called a* class invariant.

In order to define the class polynomials we make use of in our algorithm, we introduce some well known functions from number theory. All these functions are defined on the upper complex half plane which we denote by $\mathfrak{h}$.

Definition 2.2.14 *Let $\tau \in \mathfrak{h}$ and set $q = e^{2\pi i \tau}$. The function η defined by*

$$\eta(\tau) \;=\; q^{\frac{1}{24}} \cdot \prod_{n=1}^{\infty}(1 - q^n) \tag{2.14}$$

is called Dedekind's η*-function. Furthermore, the functions*

$$\mathfrak{f}(\tau) = q^{-\frac{1}{48}} \prod_{n=1}^{\infty}(1+q^{n-\frac{1}{2}}) , \tag{2.15}$$

$$\mathfrak{f}_1(\tau) = q^{-\frac{1}{48}} \prod_{n=1}^{\infty}(1-q^{n-\frac{1}{2}}) , \tag{2.16}$$

$$\mathfrak{f}_2(\tau) = \sqrt{2}q^{\frac{1}{24}} \prod_{n=1}^{\infty}(1+q^{n}) \tag{2.17}$$

are called the Weber function $\mathfrak{f}$, $\mathfrak{f}_1$, *and* $\mathfrak{f}_2$, *respectively. Finally, if we set*

$$\gamma_2(\tau) = \frac{\mathfrak{f}^{24}(\tau)-16}{\mathfrak{f}^{8}(\tau)}, \tag{2.18}$$

$$j(\tau) = \gamma_2^3(\tau), \tag{2.19}$$

then j *is called the* modular function j.

It is well known that γ_2 may be expressed as

$$\gamma_2(\tau) = \frac{\mathfrak{f}^{24}(\tau)-16}{\mathfrak{f}^{8}(\tau)} = \frac{\mathfrak{f}_1^{24}(\tau)+16}{\mathfrak{f}_1^{8}(\tau)} = \frac{\mathfrak{f}_2^{24}(\tau)+16}{\mathfrak{f}_2^{8}(\tau)} . \tag{2.20}$$

Let Δ be an imaginary quadratic discriminant. We next explain how to map a reduced representative (a,b,c) of the class group $Cl(\Delta)$ to an imaginary quadratic number. Let $Q=(a,b,c)$ be an element of $C(\Delta)$. We make use of a map which is similar to the function Φ of Theorem 2.1.14, that is we set $\tau_Q = (-b+i\sqrt{|\Delta|})/(2a)$. It is obvious that $\tau_Q \in \mathfrak{h}$. Let g denote one of the functions of Definition 2.2.14. With each $Q \in C(\Delta)$ and g we associate the complex number $g(\tau_Q)$. In addition, if Q is obvious from the context, we make use of the notation $g(\tau)$, too.

We turn to the discussion of class polynomials. We first present the *ring class polynomial*. However, before we are able to define it, we state the *First Main Theorem* of complex multiplication ([Cox89], Theorem 11.1, p.220).

Theorem 2.2.15 *Let* Δ *be an imaginary quadratic discriminant, and let* $K=\mathbb{Q}(\sqrt{\Delta})$. *In addition, let* $Q \in C(\Delta)$ *be the main form. Then* $j(\tau_Q)$ *is an algebraic integer and* $K(j(\tau_Q))$ *is the ring class field of the order* $\mathcal{O}_\Delta$.

Once, we know that $j(\tau_Q)$ is an algebraic integer for the main form, we may define its minimal polynomial.

Definition 2.2.16 *Let* Δ *be an imaginary quadratic discriminant. In addition, let* Q *be the main form of discriminant* Δ. *The minimal polynomial of* $j(\tau_Q)$ *over* $\mathbb{Q}$ *is called the* ring class polynomial. *It is denoted by* H_Δ.

We will often write H for the ring class polynomial H_Δ, if the discriminant is fixed. It is well known, that H is irreducible in $K[X]$, too. Hence, H is the minimal polynomial of $j(\tau_Q)$ over K, too. In particular, H is a polynomial of degree $h(\Delta)$. However, for H actually to be a class polynomial, we have to show that $H(j(\tau_Q)) = 0$ for all reduced representatives $Q \in C(\Delta)$. The following theorem states this fact ([Cox89], Proposition 13.2, p. 286).

Theorem 2.2.17 *Let Δ be an imaginary quadratic discriminant of class number h, and let $K = \mathbb{Q}(\sqrt{\Delta})$. In addition, let $Q_1, \ldots, Q_h$ be the reduced representatives of $C(\Delta)$. Then the ring class polynomial H_Δ is given by*

$$H_\Delta = \prod_{k=1}^{h} \left(X - j(\tau_{Q_k})\right) \ .$$

Hence if $Q \in C(\Delta)$, the complex number $j(\tau_Q)$ is a class invariant. We next introduce a class polynomial due to Atkin and Morain ([AM93]) which we denote by G. G is a polynomial associated to the cube root γ_2 of j. It turns out that G has much smaller coefficients than the ring class polynomial H. However, we have to restrict the residue of Δ mod 3.

Theorem 2.2.18 *Let Δ be an imaginary quadratic discriminant of class number h, and let $K = \mathbb{Q}(\sqrt{\Delta})$. Assume $\Delta \not\equiv 0 \bmod 3$. If we set $\zeta_3 = e^{2\pi i/3}$, then the polynomial G_Δ given by*

$$G_\Delta = \prod_{(a,b,c)\in C(\Delta)} \left(X - \zeta_3^{b(a-c+a^2c)}\gamma_2(\tau_{(a,b,c)})\right)$$

is a class polynomial.

We conclude from Theorem 2.2.18 that $\zeta_3^{b(a-c+a^2c)}\gamma_2(\tau_{(a,b,c)})$ is a class invariant if $(a,b,c) \in C(\Delta)$. Furthermore, it is easy to see that if α is a root of G_Δ, then α^3 is a root of H_Δ. Finally, we introduce a class polynomial denoted by W which is due to Yui and Zagier ([YZ97]). Again, we impose restrictions on the residue of the imaginary quadratic discriminant Δ.

Theorem 2.2.19 *Let Δ be an imaginary quadratic discriminant of class number h, and let $K = \mathbb{Q}(\sqrt{\Delta})$. In addition, assume $\Delta \not\equiv 0 \bmod 3$, $\Delta \equiv 1 \bmod 8$. Furthermore, let $\zeta_{48} = e^{2\pi i/48}$. Basing on the Weber functions of Definition 2.2.14, define a function f by*

$$f(\tau_{(a,b,c)}) := \begin{cases} \zeta_{48}^{b(a-c-ac^2)} \cdot \mathfrak{f}(\tau_{(a,b,c)}) & \text{if} \quad 2 \mid a, \ \ 2 \mid c \ , \\ (-1)^{\frac{\Delta-1}{8}} \cdot \zeta_{48}^{b(a-c-ac^2)} \cdot \mathfrak{f}_1(\tau_{(a,b,c)}) & \text{if} \quad 2 \mid a, \ \ 2 \nmid c \ , \\ (-1)^{\frac{\Delta-1}{8}} \cdot \zeta_{48}^{b(a-c+a^2c)} \cdot \mathfrak{f}_2(\tau_{(a,b,c)}) & \text{if} \quad 2 \nmid a, \ \ 2 \mid c \ . \end{cases} \tag{2.21}$$

Then the polynomial W_Δ given by

$$W_\Delta = \prod_{(a,b,c)\in C(\Delta)} \left(X - f(\tau_{(a,b,c)})\right)$$

is a class polynomial.

Again we deduce that if $\Delta \not\equiv 0 \bmod 3$, $\Delta \equiv 1 \bmod 8$, then $f(\tau_Q)$ is a class invariant. In addition, it is easy to see that $j(\tau_Q) = (f^{24}(\tau_Q) - 16)^3 / f^{24}(\tau_Q)$.

We conclude this section by estimating the size of the coefficients of the ring class polynomial H. Their order of magnitude is crucial if we want to compute the polynomial in practice, as the floating point precision in the computation depends on this magnitude. We set $q = e^{2\pi i \tau_Q}$ with $Q = (a, b, c)$. We first remark that for (a, b, c) and $(a, -b, c)$ the appropriate values of q are conjugate complex numbers. In addition, using the Fourier series of j (which we will present in Section 7.1.1), it is easy to see that $j(\tau_{(a,-b,c)}) = \overline{j(\tau_{(a,b,c)})}$. Thus computing $j(\tau_{(a,b,c)})$ yields $j(\tau_{(a,-b,c)})$ for free in this case. Hence we can restrict to the $h_p(\Delta)$ reduced representatives (a, b, c) with $b \geq 0$. Furthermore, we have $|q| = e^{-\pi\sqrt{|\Delta|}/a}$. Hence $|q|$ only depends on a. Again making use of the Fourier series of j, we deduce $|j(\tau_Q)| \approx |1/q| = e^{\pi\sqrt{|\Delta|}/a}$. Thus the constant term of H is up to sign of order of magnitude $e^{\pi\sqrt{|\Delta|}\sum_{(a,b,c)\in C(\Delta)} \frac{1}{a}}$. In most cases the constant term of H is up to sign the biggest coefficient. Hence the decimal length of the coefficients is approximately bounded by

$$\frac{\pi\sqrt{|\Delta|}}{\log 10} \cdot \sum_{(a,b,c)\in C(\Delta)} \frac{1}{a} \,. \tag{2.22}$$

Indeed, Lay and Zimmer ([LZ94]) propose to use the floating point precision F, where F is defined as

$$F = 5 + \frac{h(\Delta)}{4} + \frac{\pi\sqrt{|\Delta|}}{\log 10} \cdot \sum_{(a,b,c)\in C(\Delta)} \frac{1}{a} \,. \tag{2.23}$$

We discuss in detail the problem of choosing an appropriate floating point precision in Section 9.

2.3 Elliptic Curves

In this section we present those aspects of the theory of elliptic curves we make use of in our algorithm. First, in Section 2.3.1 we give a compact introduction to elliptic curves over a field. Next, as our generating algorithm is closely related to elliptic curves over the complex numbers $\mathbb{C}$, we discuss this theory in Section 2.3.2. Based on this theory we introduce complex multiplication in Section 2.3.3. Finally, we turn to elliptic curves over finite fields in Section 2.3.4 and discuss the requirements for their use in cryptography in Section 2.3.5. The relevant literature we cite is [BSS99], [Cox89], [Men93], and [Sil86].

2.3.1 Elliptic curves over Fields

We introduce the basic facts of elliptic curves over a general field. In our generating algorithm we are only concerned with elliptic curves over fields of characteristic different from 2 and 3. Hence in our investigation of elliptic curves we mainly restrict to this case ([Sil86], p. 50).

Definition 2.3.1 *Let F be a field with* $\mathrm{char}(F) \notin \{2; 3\}$. *The pair $(a, b) \in F^2$ is called an* elliptic curve *defined over F if $4a^3 + 27b^2 \neq 0$. The elliptic curve is denoted by E. The set of* rational points *on E over F denoted by $E(F)$ is*

$$E(F) = \{(x, y) \in F^2 : y^2 = x^3 + ax + b\} \cup \{O\} \,, \tag{2.24}$$

where O is the projective closure of the equation $y^2 = x^3 + ax + b$. The point O is called the point at infinity. *We set $O = (\infty, \infty)$.*

The equation $y^2 = x^3 + ax + b$ defined by the elliptic curve (a, b) is called the *Weierstrass equation* of (a, b). It is easy to see that the property $4a^3+27b^2 \neq 0$ ensures that the polynomial $X^3 + aX + b$ has no multiple roots over $\overline{F}$; hence the polynomial $Y^2 - X^3 - aX - b \in \overline{F}[X, Y]$ is irreducible in $\overline{F}[X, Y]$.

It is a basic fact that $E(F)$ carries a group structure with the point at infinity acting as the zero element. The binary operation of rational points in $E(F)$ is commonly denoted as an addition. Once we investigate the case $F = \mathbb{C}$ in Section 2.3.2, we will identify the addition of points in $E(\mathbb{C})$ with adding complex numbers modulo a lattice. However, it turns out that in case of a general field F the addition of points in $E(F)$ has a simple geometric interpretation, too. The addition law uses the *chord-tangent process.* First, if $P \in E(F) \setminus \{O\}$, $P = (x, y)$, we obviously have $(x, -y) \in E(F) \setminus \{O\}$, too; we set $-P := (x, -y)$. Next, let $P, Q \in E(F) \setminus \{O\}$, $P \neq Q$, and $P \neq -Q$, that is P and Q have different x-coordinates. It is easy to see that the line through P and Q intersects $E(F)$ in a third point $R \in E(F) \setminus \{O\}$. We set $P + Q := -R$. Finally, if $P' \neq O$ and $P' \neq -P'$, the tangent to $E(F)$ at P' intersects $E(F)$ in a further point $R' \in E(F) \setminus \{O\}$, and we set $2P' := -R'$. For the elliptic curve $(-1, 0)$ over $\mathbb{R}$, that is the elliptic curve with Weierstrass equation $y^2 = x^3 - x$, the chord-tangent process is shown in Figure 2.1. The formal description of this geometric interpretation is given in Theorem 2.3.2 ([Men93], Theorem 2.3, p. 18).

Theorem 2.3.2 *Let F be a field with* $\mathrm{char}(F) \notin \{2; 3\}$, *and let $E = (a, b)$ be an elliptic curve over F. We define a binary operation denoted by $+$ in $E(F)$ as follows:*

1. *$P + O = O + P = P$ for all $P \in E(F)$.*
2. *If $P = (x, y) \in E(F) \setminus \{O\}$, set $-P = (x, -y)$. In addition, set $O = -O$. Furthermore, let $P + (-P) = O$.*
3. *Let $P, Q \in E(F) \setminus \{O\}$, $P \neq -Q$, $P = (x_1, y_1)$, $Q = (x_2, y_2)$. Set $\lambda = (y_2 - y_1)/(x_2 - x_1)$ if $P \neq Q$, and $\lambda = (3x_1^2 + a)/(2y_1)$ if $P = Q$. In addition, set $P + Q = (x_3, y_3)$ where x_3 and y_3 are defined as*

$$x_3 = \lambda^2 - x_1 - x_2, \qquad y_3 = \lambda(x_1 - x_3) - y_1 \ .$$

Then $(E(F), +)$ is an Abelian group.

We next turn to maps $\phi : E(F) \to E(F)$ which respect the structure of an elliptic curve.

Definition 2.3.3 *Let $E = (a, b)$ be an elliptic curve over a field F,* $\mathrm{char}(F) \notin \{2; 3\}$. *The integral domain $\overline{F}[E] := \overline{F}[X, Y]/(Y^2 - X^3 - aX - b)$ is called the* coordinate ring of E. *The corresponding field of fractions is called the* function field of E, *denoted by $\overline{F}(E)$. An element of $\overline{F}(E)$ is called a* rational function of E. *The rings $F'[E]$ and $F'(E)$ for an extension field F'/F comprise all elements of $\overline{F}[E]$ and $\overline{F}(E)$, respectively, which are invariant under the action of* $\mathrm{Gal}(\overline{F} : F')$. *A rational function r is defined over an extension field F'/F, if there is a representation $r = f/g$ with $f, g \in F'[E]$.*

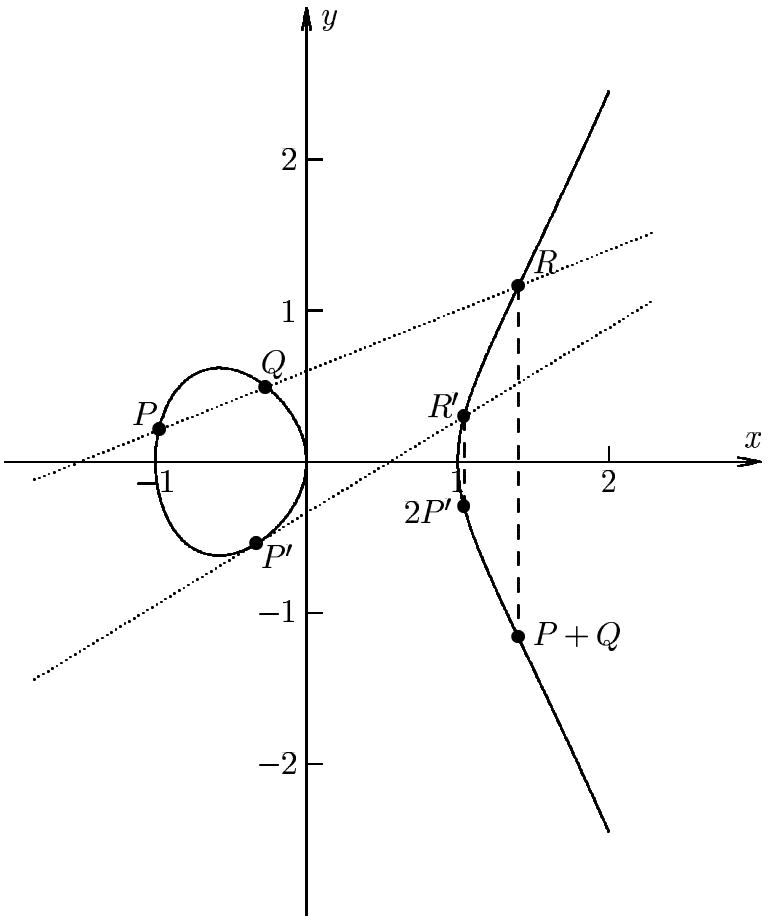


Figure 2.1: Addition law on the elliptic curve $E = (-1, 0)$ over $\mathbb{R}$.

Let E be an elliptic curve defined over a field F. Let $P \in E(F)$ and $r \in F(E)$. We follow [Men93], p. 29-31, to define $r(P)$. First, let $P \neq O$. Then r is said to be *defined at* P if there are elements $g, h \in F[E]$ with $r = g/h$ and $h(P) \neq 0$. If this is true, then we set $r(P) = g(P)/h(P)$. It is easy to see that this definition is well-defined. If r is not defined at P we set $r(P) = \infty$. Next, let $P = O$. For $g \in F[E]$ there is a unique representation $g = v + wY$, where both $v, w \in F[X]$. As usual we set $\deg(g) = \max(2\deg(v), 3 + 2\deg(w))$, that is X and Y are of weight 2 and 3, respectively. Now let $r = g/h$, where both g and h have the form $v + wY$ from above. Then $r(O) = 0$ or $r(O) = \infty$ depending on whether $\deg(g) < \deg(h)$ or $\deg(g) > \deg(h)$, respectively. Otherwise we set $r(O) = c/d$ where c and d are the coefficients of the highest terms of g and h, respectively. We are now able to define an isogeny (see for instance [BSS99], p. 44).

Definition 2.3.4 *Let $E = (a, b)$ and $E' = (a', b')$ be elliptic curves over a field F. A* non-constant isogeny *from E to E' is a pair of rational functions $(r, s) \in \overline{F}(E)^2$ with $s^2 = r^3 + a'r + b'$ and $(r(O), s(O)) = (\infty, \infty)$. In addition, the map $\phi_0 : P \mapsto O$ for all $P \in E(\overline{F})$ is called the* constant isogeny. *An isogeny is defined over an extension field F' of F, if r and s are both defined over F'.*

It is a basic fact that any isogeny $\phi : E \to E$ is a homomorphism of $E(\overline{F})$ to itself, i.e. an endomorphism of $E(\overline{F})$. Hence if the isogeny ϕ is defined over F' it gives an endomorphism of $E(F')$. However, an isogeny has stronger properties, as it respects the analytic structure of $E(\overline{F})$. This will be rather important when considering elliptic curves over $\mathbb{C}$. Once we know that isogenies respect the addition law of $E(\overline{F})$, we may use the composition of isogenies to define the endomorphism ring of E ([Sil86], p. 71).

Definition 2.3.5 *Let F be a field and $E = (a, b)$ be an elliptic curve defined over F. The set of all isogenies $E \to E$ is called the* endomorphism ring of E. *It is denoted by* $\mathrm{End}(E)$. *In addition, the set of all isogenies $(r, s) : E \to E$, where both r and s are defined over F is called the* endomorphism ring of E over F. *It is denoted by* $\mathrm{End}_F(E)$.

We next consider the map $[m] : P \mapsto [m] \cdot P$ for $m \in \mathbb{Z}$ and $P \in E(F)$. The formulae in Theorem 2.3.2 show that this map is in both $\mathrm{End}_F(E)$ and $\mathrm{End}(E)$. Hence both $\mathrm{End}_F(E)$ and $\mathrm{End}(E)$ contain a subring isomorphic to $\mathbb{Z}$. The image of P under the map $[m]$ is denoted by $[m]P$. We next define the central term of *complex multiplication.* ([Sil86], p. 73).

Definition 2.3.6 *Let $E = (a, b)$ be an elliptic curve defined over a field F. The elliptic curve is said to have* complex multiplication *if the endomorphism ring* $\mathrm{End}(E)$ *is strictly larger than* $\mathbb{Z}$.

We define two important terms related to a given elliptic curve E : The discriminant Δ and the j-invariant of an elliptic curve ([Men93], p. 19).

Definition 2.3.7 *Let F be a field with* $\mathrm{char}(F) \notin \{2; 3\}$, *and let $E = (a, b)$ be an elliptic curve defined over F. The* discriminant *of E is defined as $\Delta(E) = -16(4a^3 + 27b^2)$. In addition, the number $j(E) = 1728 \cdot 4a^3/(4a^3 + 27b^2)$ is called the* j-invariant *of E.*

As $\mathrm{char}(F) \neq 2$, we obviously have $\Delta(E) \neq 0$ for an elliptic curve E over F. The name j-invariant comes from the invariance of this quantity under isomorphisms. Isomorphic elliptic curves have the same j-invariant ([Hus87], p. 68). Conversely, if the field F is algebraically closed, then two elliptic curves with the same j-invariant are isomorphic ([Men93], Theorem 2.5, p. 19).

We show that every element of a field F, $\mathrm{char}(F) \notin \{2; 3\}$, is the j-invariant of some elliptic curve over F. First, assume $j_0 \in F$, $j_0 \notin \{0; 1728\}$. In addition, set

$$\kappa = j_0/(1728 - j_0) \qquad \text{and} \qquad E = (3\kappa, 2\kappa)\ . \tag{2.25}$$

Then we have $\Delta(E) = -1728\kappa^2(\kappa + 1)$. Obviously $\kappa = -1$ is equivalent to $1728 = 0$, which contradicts $\mathrm{char}(F) \neq 2$. In addition, $j_0 \neq 0$ induces $\kappa \neq 0$. Thus we have $\Delta(E) \neq 0$. Hence E is an elliptic curve over F. Furthermore, we determine the j-invariant of E to

$$\begin{aligned} j(E) &= 1728 \cdot \tfrac{4a^3}{4a^3+27b^2} &&= 1728 \cdot \tfrac{1}{1+\frac{27b^2}{4a^3}} \\ &= 1728 \cdot \tfrac{1}{1+\frac{27\cdot 4\kappa^2}{4\cdot 27\kappa^3}} &&= 1728 \cdot \tfrac{1}{1+\frac{1}{\kappa}} \\ &= 1728 \cdot \tfrac{1}{1+\frac{1728-j_0}{j_0}} &&= 1728 \cdot \tfrac{j_0}{j_0+1728-j_0} \\ &= j_0. \end{aligned}$$

Second, it is easy to see that the elliptic curves $(0,1)$ and $(1,0)$ have j-invariant 0 and 1728, respectively. Thus we have proven the following proposition.

Proposition 2.3.8 *Let F be a field,* $\mathrm{char}(F) \notin \{2;3\}$, *and let $j_0 \in F$. There is an elliptic curve E defined over F with $j(E) = j_0$.*

2.3.2 Elliptic curves over $\mathbb{C}$

In order to explain the theory of complex multiplication, we have to discuss elliptic curves over the complex numbers $\mathbb{C}$. It is the aim of this section to present the basic facts in this context. We will show that elliptic curves over $\mathbb{C}$ and complex lattices are essentially the same object. This identification yields a natural explanation of the addition law of points in the group of rational points of an elliptic curve over $\mathbb{C}$. Furthermore, basing on these investigations we will be able to introduce the theory of complex multiplication in Section 2.3.3. We first introduce the fundamental term of a lattice ([Cox89], p. 200).

Definition 2.3.9 *Let $\omega_1, \omega_2 \in \mathbb{C}$, linearly independent over $\mathbb{R}$. The additive subgroup $\mathcal{L} = \omega_1\mathbb{Z} + \omega_2\mathbb{Z}$ is called a* lattice in $\mathbb{C}$. *We also write $\mathcal{L} = [\omega_1, \omega_2]$. Two lattices $\mathcal{L}_1$ and $\mathcal{L}_2$ are* homothetic *if there is a $\lambda \in \mathbb{C}^\times$ with $\mathcal{L}_1 = \lambda\mathcal{L}_2$.*

For instance, fractional ideals $\mathfrak{a}$ of imaginary quadratic orders $\mathcal{O}_\Delta$ are lattices, as already discussed in Section 2.1.1. In addition, homothety of lattices induces an equivalence relation on the set of lattices. We define the modular function j on a homothety class of lattices as follows: Let $\mathcal{L} := [\omega_1, \omega_2]$ be a lattice in $\mathbb{C}$. Obviously, the lattice $[1, \omega_2/\omega_1]$ is homothetic to $\mathcal{L}$, where we may assume without loss of generality that $\omega_2/\omega_1 \in \mathfrak{h}$. We set $j(\mathcal{L}) := j(\omega_2/\omega_1)$. To justify this definition we remark that the value of $j(\mathcal{L})$ is independent of the choice of the representant $\mathcal{L}$ and the generators ω_1 and ω_2. Once, we have a lattice $\mathcal{L}$ in $\mathbb{C}$, we define Eisenstein series related to $\mathcal{L}$ ([Kob93], p. 109).

Definition 2.3.10 *Let $\mathcal{L} = [\omega_1, \omega_2]$ be a lattice in $\mathbb{C}$. The complex numbers*

$$G_2(\mathcal{L}) := \sum_{\omega \in \mathcal{L}\setminus\{0\}} \frac{1}{\omega^4} \qquad \text{and} \qquad G_3(\mathcal{L}) := \sum_{\omega \in \mathcal{L}\setminus\{0\}} \frac{1}{\omega^6}$$

are called Eisenstein series *for the lattice $\mathcal{L}$. In addition, we set $g_2(\mathcal{L}) := 60G_2(\mathcal{L})$ and $g_3(\mathcal{L}) := 140G_3(\mathcal{L})$.*

For each lattice there is a class of meromorphic functions on $\mathbb{C}$ defined as follows ([Cox89], p. 200).

Definition 2.3.11 *Let $\mathcal{L} = [\omega_1, \omega_2]$ be a lattice in $\mathbb{C}$. In addition, let $f : \mathbb{C} \to \widehat{\mathbb{C}}$ be a meromorphic function on $\mathbb{C}$. Furthermore, assume that f is periodic with respect to $\mathcal{L}$, that is $f(z) = f(z + \omega)$ for all $z \in \mathbb{C}$, $\omega \in \mathcal{L}$. Then f is called an* elliptic function *for the lattice $\mathcal{L}$.*

We remark that an elliptic function is a doubly periodic meromorphic function. We next introduce the most important elliptic function, the *Weierstrass $\wp$-function.* It is common to define the $\wp$-function as an infinite sum ([Cox89], p. 200).

Definition 2.3.12 *Let $\mathcal{L}$ be a lattice in $\mathbb{C}$. The function $\wp : \mathbb{C} \to \widehat{\mathbb{C}}$ defined as*

$$\wp(z;\mathcal{L}): \quad z \mapsto \begin{cases} \frac{1}{z^2} + \sum_{\omega \in \mathcal{L}\setminus\{0\}} \left(\frac{1}{(z-\omega)^2} - \frac{1}{\omega^2}\right), & if \quad z \notin \mathcal{L}\,, \\ \infty \qquad\qquad\qquad\qquad\qquad\quad , & if \quad z \in \mathcal{L} \end{cases}$$

is called the Weierstrass $\wp$-function *for the lattice $\mathcal{L}$.*

If we work with a fixed lattice, we will usually write $\wp(z)$ instead of $\wp(z;\mathcal{L})$. We next cite a theorem stating that the $\wp$-function actually is an elliptic function. In addition, $\wp$ satisfies a differential equation which is very important to link lattices to elliptic curves over $\mathbb{C}$ ([Cox89], Theorem 10.1, p. 200; [Cox89], Proposition 10.7, p. 206).

Theorem 2.3.13 *Let $\wp$ be the Weierstrass $\wp$-function for the lattice $\mathcal{L}$.*
1. *$\wp(z)$ is an elliptic function for $\mathcal{L}$ whose singularities consist of double poles at the points of $\mathcal{L}$.*
2. *$\wp(z)$ satisfies the differential equation*

$$\wp'(z)^2 = 4\wp(z)^3 - g_2(\mathcal{L})\wp(z) - g_3(\mathcal{L})\,. \tag{2.26}$$

3. *We have $g_2(\mathcal{L})^3 - 27g_3(\mathcal{L})^2 \neq 0$.*

From Equation (2.26) it is easy to see that $\wp$ yields an elliptic curve; more precisely, dividing Equation (2.26) by 4, defining $(a,b) := (-g_2(\mathcal{L})/4, -g_3(\mathcal{L})/4)$, and scaling $(x,y) := (\wp(z;\mathcal{L}), \wp'(z;\mathcal{L})/2)$ gives an equation of the form $y^2 = x^3 + ax + b$. In addition, property 3 of Theorem 2.3.13 shows $-16(4a^3 + 27b^2) = g_2(\mathcal{L})^3 - 27g_3(\mathcal{L})^2 \neq 0$. Hence $E := (a,b)$ is an elliptic curve over $\mathbb{C}$ with corresponding Weierstrass equation $y^2 = x^3 + ax + b$.

We get a map $\mathbb{C} \to E(\mathbb{C})$ as follows: If $z \in \mathbb{C} \setminus \mathcal{L}$, we set $z \mapsto (\wp(z;\mathcal{L}), \wp'(z;\mathcal{L})/2)$. In case of $z \in \mathcal{L}$, we set $z \mapsto O$. As both $\wp$ and $\wp'$ are periodic for $\mathcal{L}$, this yields a well-defined map from $\mathbb{C}/\mathcal{L} \to E(\mathbb{C})$. We denote this map by $\mathcal{H}$. It is a fundamental fact that $\mathcal{H}$ is an isomorphism. Thus we can add two points $P, Q \in E(\mathbb{C})$ by $P + Q := \mathcal{H}(\mathcal{H}^{-1}(P) + \mathcal{H}^{-1}(Q))$, where the addition in $\mathbb{C}/\mathcal{L}$ is simply the addition in $\mathbb{C}$ modulo $\mathcal{L}$. Indeed, this addition may be described in terms of the formulae in Theorem 2.3.2.

We have previously discussed that any lattice in $\mathbb{C}$ yields an elliptic curve over $\mathbb{C}$. We next show that the converse is true, too ([Cox89], Proposition 14.3, p. 309).

Theorem 2.3.14 *Let $E = (a,b)$ be an elliptic curve over $\mathbb{C}$. Then there is a unique lattice $\mathcal{L} \subset \mathbb{C}$ such that $a = -g_2(\mathcal{L})/4$ and $b = -g_3(\mathcal{L})/4$.*

We conclude that lattices in $\mathbb{C}$ and elliptic curves over $\mathbb{C}$ are two sides of the same coin. We make use of this close relationship in the following section to introduce complex multiplication.

2.3.3 Complex Multiplication

In this section we sketch the theory of *complex multiplication.* The central object in this theory is the endomorphism ring of an elliptic curve E. In Section 2.3.1 we already stated that $\mathrm{End}(E)$ contains a subring isomorphic to $\mathbb{Z}$. However, in case of elliptic curves over $\mathbb{C}$ the endomorphism ring is well known ([Cox89], p. 312).

Theorem 2.3.15 *Let E be an elliptic curve over $\mathbb{C}$, and let $\mathcal{L}$ be the corresponding lattice. Then* $\mathrm{End}(E) \simeq \{\alpha \in \mathbb{C} : \alpha\mathcal{L} \subset \mathcal{L}\}$.

Thus the problem of finding endomorphisms of E is reduced to the problem of finding all complex numbers α which leave the lattice $\mathcal{L}$ corresponding to E invariant under multiplication. The next theorem demonstrates the close relationship between endomorphism rings of elliptic curves over $\mathbb{C}$ and imaginary quadratic orders ([Cox89], Theorem 10.14, p. 209).

Theorem 2.3.16 *Let $\mathcal{L}$ be a lattice in $\mathbb{C}$. For a number $\alpha \in \mathbb{C} \setminus \mathbb{Z}$ the following statements are equivalent:*
1. *The lattice $\mathcal{L}$ is invariant under multiplication with α, i.e. $\alpha\mathcal{L} \subset \mathcal{L}$.*
2. *There exists an order $\mathcal{O}_\Delta$ in an imaginary quadratic field K such that $\alpha \in \mathcal{O}_\Delta$ and $\mathcal{L}$ is homothetic to an invertible fractional ideal of $\mathcal{O}_\Delta$.*

Theorem 2.3.16 constitutes a fundamental result for our generating algorithm as it relates elliptic curves to imaginary quadratic orders as follows: Let $\mathcal{O}$ be an imaginary quadratic order. As usual we identify an elliptic curve E over $\mathbb{C}$ with the corresponding lattice $\mathcal{L}$. According to Theorems 2.3.16 and 2.3.15 E has $\mathcal{O}$ as its endomorphism ring if and only if $\mathcal{L}$ is homothetic to an invertible fractional ideal of $\mathcal{O}$. Hence in order to study elliptic curves over $\mathbb{C}$ with $\mathrm{End}(E) \simeq \mathcal{O}$ it is sufficient to take invertible fractional ideals of $\mathcal{O}$ into account. In addition, it is easy to see that two equivalent fractional ideals of $\mathcal{O}$ determine isogenous elliptic curves; indeed, the isogenous curves are isomorphic. However, as equivalent ideals determine the same ideal class in $Cl(\mathcal{O})$ we state the following (a variant of it may be found in [Cox89], Corollary 10.20, p. 212).

Theorem 2.3.17 *Let $\mathcal{O}$ be an imaginary quadratic order. In addition, let $I(\mathcal{O})$ be the group of invertible fractional ideals of $\mathcal{O}$. In addition, let $\mathcal{C}$ be the set of isomorphy classes of elliptic curves E over $\mathbb{C}$ with* $\mathrm{End}(E) \simeq \mathcal{O}$. *Define a map $\rho : Cl(\mathcal{O}) \to \mathcal{C}$ which sends an ideal class $\mathfrak{a}P(\mathcal{O})$ to the isomorphy class represented by the elliptic curve corresponding to $\mathfrak{a}$. Then ρ is bijective. In addition, we have $j(\mathfrak{a}) = j(E)$, where $j(\mathfrak{a})$ denotes the modular function j evaluated for the homothety class of $\mathfrak{a}$ and $j(E)$ denotes the j-invariant of an elliptic curve in the class $\rho(\mathfrak{a})$.*

We remark that the value $j(E)$ is independent of the choice of the elliptic curve in $\rho(\mathfrak{a})$. Next we relate elliptic curves over $\mathbb{C}$ having $\mathcal{O}$ as their endomorphism ring to the ring class polynomial H. If L denotes the ring class field of $\mathcal{O}$, we denote by $\mathcal{E}$ an elliptic curve defined over L. We stated in Proposition 2.3.8 that for all $j_0 \in L$ there exists an elliptic curve $\mathcal{E}$ defined over L with $j(\mathcal{E}) = j_0$. In addition, if $j_0 \notin \{0; 1728\}$, the relation between j_0 and $\mathcal{E}$ is given by Equation (2.25). Furthermore, Theorem 2.2.17 shows that if $\mathfrak{a}$ is a fractional ideal of $\mathcal{O}$, we have $K(j(\mathfrak{a}P(\mathcal{O}))) = L$, where K is the field of fractions of $\mathcal{O}$. In addition, Theorem 2.2.17 states that all values $j(\mathfrak{a}P(\mathcal{O}))$ are conjugate.

It is well known that $j(\mathcal{O}_{-3}) = 0$ and $j(\mathcal{O}_{-4}) = 1728$ (for instance, see [Cox89], p. 261). In addition, if $\Delta < -4$ and $\mathfrak{a}$ is a fractional ideal of $\mathcal{O}_\Delta$, then $j(\mathfrak{a}P(\mathcal{O}_\Delta)) \notin \{0; 1728\}$ follows. Hence, if $\Delta < -4$, using Theorem 2.3.17, we deduce the following: Let E be an elliptic curve defined over $\mathbb{C}$ and having $\mathcal{O}_\Delta$ as its endomorphism ring. In addition, let $\mathfrak{a}_i P(\mathcal{O}_\Delta)$, $1 \leq i \leq h(\Delta)$, be all elements of the ideal class group $Cl(\mathcal{O}_\Delta)$. Let k be any

element of $\{1,\ldots,h(\Delta)\}$, and let $\mathcal{E}$ be the elliptic curve over L defined by Equation (2.25) with $j_0 = j(\mathfrak{a}_k P(\mathcal{O}_\Delta))$. Then the lattices corresponding to E and $\mathcal{E}$ are homothetic. Thus up to homothety any elliptic curve E over $\mathbb{C}$ with $\mathrm{End}(E) \simeq \mathcal{O}_\Delta$ is already defined over L.

We next fix an index $k \in \{1,\ldots,h(\Delta)\}$. Again let E be defined by Equation (2.25) with $j_0 = j(\mathfrak{a}_k P(\mathcal{O}_\Delta))$. We claim that E is not defined over a proper subfield of $\mathbb{Q}(j_0)$. Hence let $E = (a,b)$ be defined over $F \subsetneq \mathbb{Q}(j_0)$. Then by Definition 2.3.7 we have $j_0 = 1728 \cdot 4a^3/(4a^3 + 27b^2) \in F$, hence $[\mathbb{Q}(j_0) : \mathbb{Q}] < h(\Delta)$, a contradiction.

2.3.4 Elliptic curves over Finite Fields

We next introduce the relevant theory of elliptic curves over finite fields. Let $E = (a,b)$ be an elliptic curve defined over the finite field $\mathbb{F}_q$, where $q = p^n$ is a prime power. If $E(\mathbb{F}_q)$ is suitable for use in cryptography the requirements of Section 2.3.5 show that $\mathrm{End}(E)$ is an imaginary quadratic order $\mathcal{O}$. The central theorem of this section states that there is an elliptic curve $\mathcal{E} = (\alpha,\beta)$ defined over the ring class field L of $\mathcal{O}$ and a prime $\mathfrak{P} \subset \mathcal{O}_L$ of inertial degree n over p with $(a,b) = (\alpha \bmod \mathfrak{P}, \beta \bmod \mathfrak{P})$. The facts of this section build the underlying theory of our generating algorithm `cryptoCurve`.

We first state the fundamental theorem concerning the cardinality of the group of rational points of an elliptic curve over a finite field. Let $\mathbb{F}_q$ be a finite field with $\mathrm{char}(\mathbb{F}_q) \geq 5$, and let $E = (a,b)$ be an elliptic curve defined over $\mathbb{F}_q$. The Weierstrass equation $y^2 = x^3 + ax + b$ shows $|E(\mathbb{F}_q)| \leq 2q + 1$. However, there is a much better bound due to Hasse, who proved a conjecture of Artin. We remark that an elliptic curve over a field of characteristic 2 or 3 may be described in terms of a similar Weierstrass equation. Hence the following theorem of Hasse is true even if $\mathrm{char}(\mathbb{F}_q) \in \{2;3\}$.

Theorem 2.3.18 (Hasse) *Let p be a prime, and let $q = p^n$. Let E be an elliptic curve over $\mathbb{F}_q$. Then $||E(\mathbb{F}_q)| - (q+1)| \leq 2\sqrt{q}$.*

The theorem of Hasse states that there is a $t \in \mathbb{Z}$, $|t| \leq 2\sqrt{q}$ with $|E(\mathbb{F}_q)| = q + 1 - t$. The integer t is called the *trace of E over $\mathbb{F}_q$*. It turns out that elliptic curves over finite fields come in two categories, *ordinary* and *supersingular*, as determined by their endomorphism rings (see [Cox89], Theorem 14.14, p. 316).

Definition and Theorem 2.3.19 *Let E be an elliptic curve over a finite field $\mathbb{F}_q$. The endomorphism ring* $\mathrm{End}(E)$ *is either an imaginary quadratic order or an order in a quaternion algebra. In the first case, E is called* ordinary *or* non-supersingular*. In the second case, E is said to be* supersingular.

Later on we will see that only ordinary elliptic curves are of interest in cryptographic applications. There are a lot of equivalent criteria for E to be supersingular. However, once we know the cardinality of an elliptic curve over a finite prime field, it is easy to decide to which category the curve belongs to ([Cox89], Proposition 14.15, p. 316).

Theorem 2.3.20 *Let $p \geq 5$ be a prime, and E be an elliptic curve over $\mathbb{F}_p$. Then E is supersingular if and only if $|E(\mathbb{F}_p)| = p + 1$.*

Equivalent to Theorem 2.3.20 is the statement that an elliptic curve over a finite prime field of characteristic at least 5 is supersingular if and only if its trace t is equal to 0.

In Definition 2.3.4 we introduced isogenies as maps between elliptic curves. We stated that isogenies respect the structure of an elliptic curve. In case of finite fields, the structure of isomorphisms is as follows ([Len87], 1.3, p. 653). In addition, we introduce the important term of *twists* of elliptic curves.

Definition and Theorem 2.3.21 *Let $E = (a, b)$ and $E' = (a', b')$ be elliptic curves defined over $\mathbb{F}_q$, $\mathrm{char}(\mathbb{F}_q) \geq 5$.*
1. *Let $\phi : E \to E'$ be an isogeny defined over $\mathbb{F}_q$. In addition, assume $E(\mathbb{F}_q)$ and $E'(\mathbb{F}_q)$ to be isomorphic. Then there is an $\alpha \in \mathbb{F}_q^\times$ with $a' = a \cdot \alpha^4$ and $b' = b \cdot \alpha^6$. The curves are called* $\mathbb{F}_q$-isomorphic. *We denote $\mathbb{F}_q$-isomorphic elliptic curves by $E/\mathbb{F}_q \simeq E'/\mathbb{F}_q$.*
2. *The curves E and E' are said to be* $\mathbb{F}_q$-twisted *iff there exists a non-square $\beta \in \mathbb{F}_q^\times$ with $a' = a \cdot \beta^2$ and $b' = b \cdot \beta^3$. If E' is $\mathbb{F}_q$-twisted to E we say that E' is a twist of E over $\mathbb{F}_q$.*

We next prove some well-known results. We will make use of these results in our algorithm `cryptoCurve`.

Theorem 2.3.22 *Let $E = (a, b)$ be an elliptic curve defined over $\mathbb{F}_q$, $\mathrm{char}(\mathbb{F}_q) \geq 5$. Let E' be a twist of E over $\mathbb{F}_q$. In addition, if $b = 0$ we assume $q \equiv 1 \bmod 4$. Then we have:*
1. $E(\mathbb{F}_q) \not\simeq E'(\mathbb{F}_q)$.
2. $E(\mathbb{F}_{q^2}) \simeq E'(\mathbb{F}_{q^2})$.
3. $|E(\mathbb{F}_q)| + |E'(\mathbb{F}_q)| = 2q + 2$.

Proof: As E' is a twist of E over $\mathbb{F}_q$ there exists a non-square $\beta \in \mathbb{F}_q^\times$ with $E' = (a \cdot \beta^2, b \cdot \beta^3)$.
1. Assume, that $E(\mathbb{F}_q) \simeq E'(\mathbb{F}_q)$ holds. Hence there is a field element $\alpha \in \mathbb{F}_q^\times$ with $E' = (a \cdot \alpha^4, b \cdot \alpha^6)$. If $b \neq 0$ we have $\beta^3 = \alpha^6$, hence $-1 = (\beta^3)^{(q-1)/2} = (\alpha^6)^{(q-1)/2} = 1$, which is a contradiction. If $b = 0$ then $\beta^2 = \alpha^4$ follows. Let ξ be a generator of $\mathbb{F}_q^\times$. As β is a non-square, there is a $1 \leq s < (q-1)/2$ with $\beta = \xi^{2s+1}$. Furthermore, assume $\alpha = \xi^u$. Hence we have $\xi^{4s+2} = \xi^{4u}$ yielding $4s + 2 \equiv 4u \bmod q - 1$. Thus we deduce $q - 1 = 4 \cdot v \mid 4(s - u) + 2$ for an integer v. However, this is impossible.
2. Let $\alpha \in \mathbb{F}_{q^2}$ be a square root of β. Thus $E' = (a \cdot \alpha^4, b \cdot \alpha^6)$ follows, and we have $E(\mathbb{F}_{q^2}) \simeq E'(\mathbb{F}_{q^2})$.
3. Let $f = x^3 + ax + b$ and $f' = x^3 + a\beta^2 x + b\beta^3$. We show that for each $x_0 \in \mathbb{F}_q$ there are exactly two points in either $E(\mathbb{F}_q)$ or $E'(\mathbb{F}_q)$ having x-coordinate x_0 or βx_0, respectively. If $f(x_0) = 0$, then $(x_0, 0) \in E(\mathbb{F}_q)$ and $(\beta x_0, 0) \in E'(\mathbb{F}_q)$ follows. If $f(x_0)$ is a square in $\mathbb{F}_q^\times$, then the two different points $(x_0, \pm\sqrt{f(x_0)})$ are in $E(\mathbb{F}_q)$. However, obviously $f'(\beta x_0)$ is a non-square in $\mathbb{F}_q^\times$. Hence there is no point in $E'(\mathbb{F}_q)$ having x-coordinate βx_0. Finally, let $f(x_0)$ be a non-square. In this case there is no point in $E(\mathbb{F}_q)$ having x-coordinate x_0. However, $f'(\beta x_0)$ is a square in $\mathbb{F}_q^\times$. Thus the two different points $(\beta x_0, \pm\sqrt{f'(\beta x_0)})$ are in $E'(\mathbb{F}_q)$. □

Finally, we turn to the relationship between ordinary elliptic curves over finite fields and elliptic curves over number fields. As usual, let $\mathcal{O}$ be an imaginary quadratic order and L its ring class field. If E is an elliptic curve over $\mathbb{F}_q$ with $\mathrm{End}(E) \simeq \mathcal{O}$, then according to Definition 2.3.19 E is ordinary. In addition, in Section 2.3.3 we stated that any elliptic curve over $\mathbb{C}$

having $\mathcal{O}$ as endomorphism ring may be represented up to homothety of the corresponding lattice by an elliptic curve $\mathcal{E}$ defined over L. Thus it is not surprising that there exists a close relationship between ordinary elliptic curves E over $\mathbb{F}_q$ and elliptic curves $\mathcal{E}$ over L both having $\mathcal{O}$ as their endomorphism ring. Before we can state the fundamental theorem due to Deuring, we have to explain how to reduce elliptic curves $\mathcal{E}$ over number fields to elliptic curves over finite fields.

Let $\mathcal{E} = (\alpha, \beta)$ be an elliptic curve over a global number field F. In addition, let $\mathfrak{P}$ be a prime of F, i.e. a prime ideal $\mathfrak{P} \neq (0)$ of $\mathcal{O}_F$. Thus, according to Theorem 2.2.4, $\mathcal{O}_F/\mathfrak{P} \simeq \mathbb{F}_q$ for some prime power $q = p^f$. We assume $p \geq 5$. In addition, as F is the field of fractions of $\mathcal{O}_F$, there are elements $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathcal{O}_F$ with $\alpha = \alpha_1/\alpha_2$ and $\beta = \beta_1/\beta_2$. If both $\alpha_2 \notin \mathfrak{P}$ and $\beta_2 \notin \mathfrak{P}$, then $a := \alpha_1 \cdot \alpha_2^{-1} \bmod \mathfrak{P}$ and $b := \beta_1 \cdot \beta_2^{-1} \bmod \mathfrak{P}$ are in $\mathbb{F}_q$. Furthermore, if $4a^3 + 27b^2 \neq 0$ we say that $\mathcal{E}$ has *good reduction* modulo $\mathfrak{P}$. We set $E = (a, b)$ and say that E is the reduction of $\mathcal{E}$ modulo $\mathfrak{P}$.

The following theorem due to Deuring is the main theorem of our generating algorithm `cryptoCurve` ([LZ94], Theorem 4, p. 251).

Theorem 2.3.23 (Deuring) *Let $\mathcal{O}$ be an imaginary quadratic order of conductor f, K its field of fractions, and L its ring class field. In addition, let p be a rational prime which splits completely in K, and let $\mathfrak{P}$ be a prime of L containing $p\,\mathcal{O}_L$. Furthermore, let $f_{\mathfrak{P}}$ denote the inertial degree of $\mathfrak{P}$ over $p\mathbb{Z}$. We assume $f \notin \mathfrak{P}$. Let $\mathcal{E}$ be an elliptic curve defined over L with $\mathrm{End}(\mathcal{E}) = \mathcal{O}$ and having good reduction modulo $\mathfrak{P}$. Denote by E the reduction of $\mathcal{E}$ modulo $\mathfrak{P}$. Then there is an element $\pi \in \mathcal{O} \setminus p\,\mathcal{O}$ with*

$$\pi \cdot \overline{\pi} = p^{f_{\mathfrak{P}}}, \qquad |E(\mathbb{F}_{p^{f_{\mathfrak{P}}}})| = p^{f_{\mathfrak{P}}} + 1 - (\pi + \overline{\pi}). \tag{2.27}$$

In addition, $\mathrm{End}(E) \simeq \mathcal{O}$. Finally, every elliptic curve E over $\mathbb{F}_{p^{f_{\mathfrak{P}}}}$ with $\mathrm{End}(E) \simeq \mathcal{O}$ arises in this way.

We discuss the relevance of Theorem 2.3.23 to algorithm `cryptoCurve`. The main observation concerns the ring class polynomial H. Let p and $\mathfrak{P}$ be primes as in Theorem 2.3.23, and let $\mathcal{E}$ be an elliptic curve with $\mathrm{End}(\mathcal{E}) = \mathcal{O}$. From the end of Section 2.3.3 (see for instance Theorem 2.3.17 and the subsequent discussion) we know that $H(j(\mathcal{E})) = 0$. In addition, $j(E) := j(\mathcal{E}) \bmod \mathfrak{P}$ is a root of $H \bmod p$. It is a basic fact of class field theory that $j(E) \in \mathbb{F}_{p^{f_{\mathfrak{P}}}}$, but $j(E) \notin \mathbb{F}_q$ for any proper subfield $\mathbb{F}_q$ of $\mathbb{F}_{p^{f_{\mathfrak{P}}}}$. Thus $H \bmod p$ splits into different irreducible factors of degree $f_{\mathfrak{P}}$. Conversely, let j_p be a root of $H \bmod p$. In addition, let $\Delta < -4$. As discussed in Equation (2.25) of Section 2.3.1 we set $\kappa_p = j_p/(1728 - j_p)$ and $E = (3\kappa_p, 2\kappa_p)$. Then using the last statement of Theorem 2.3.23 there is an element $\pi \in \mathrm{End}(E)$ such that $|E(\mathbb{F}_{p^{f_{\mathfrak{P}}}})| = p^{f_{\mathfrak{P}}} + 1 - (\pi + \overline{\pi})$.

We are now able to explain the significance of Theorem 2.3.23 for our algorithm `cryptoCurve`. In `cryptoCurve` we are mostly concerned with maximal orders. Hence it follows $f = 1$ for the conductor f of $\mathcal{O}$. Thus we conclude $f \notin \mathfrak{P}$ for any prime $\mathfrak{P}$ of L. Furthermore, the ring class field actually is the Hilbert class field. In addition, we have to generate elliptic curves over finite prime fields, that is the inertial degree $f_{\mathfrak{P}}$ is assumed to be 1, too. We make use of the following theorem to decide whether $f_{\mathfrak{P}} = 1$ (a variant of it may be found in [Cox89], Corollary 5.25, p. 109).

Theorem 2.3.24 *Let p be a rational prime and $\mathcal{O}_K$ be a maximal imaginary quadratic order. In addition, assume $p\mathcal{O}_K = \mathfrak{p}\overline{\mathfrak{p}}$ with $\mathfrak{p} \neq \overline{\mathfrak{p}}$. Finally, denote by $f_{\mathfrak{P}}$ the inertial degree of a prime $\mathfrak{P}$ of the Hilbert class field of $\mathcal{O}_K$ over p. Then $f_{\mathfrak{P}} = 1$ if and only if $\mathfrak{p}$ is principal.*

Hence Theorem 2.3.24 shows that for a prime p which splits completely in K the reduced curve of Theorem 2.3.23 is defined over $\mathbb{F}_p$ if and only if there is a $\pi \in \mathcal{O}_K$ with $p = \pi\overline{\pi}$. We refer to an equation of the form $p = \pi\overline{\pi}$ as a *norm equation*. Our efficient algorithms to solve a norm equation in Section 4 will make use of our investigations of Section 2.1.3. Let $\pi \in \mathcal{O}_K$ with $p = \pi\overline{\pi}$. In Proposition 2.1.19 we proved that there are integers $(t, y) \in \mathbb{Z}^2$ with $4p = t^2 - \Delta y^2$. It is easy to see that t is the trace of π, that is $\pi + \overline{\pi} = t$. Hence, using Theorem 2.3.23 and a root of $H \bmod p$ as discussed above yields a curve of order $p + 1 - t$.

We conclude this section with an explanation of mapping roots of a class polynomial to roots of the ring class polynomial. First, let $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$. Then the polynomial W_Δ from Theorem 2.2.19 is a class polynomial. If p is a prime as in the theorem of Deuring, then the splitting behavior of $W_\Delta \bmod p$ and $H_\Delta \bmod p$ is the same. If w_p denotes a root of $W_\Delta \bmod p$, then $(w_p^{24} - 16)^3 / w_p^{24}$ is a root of $H_\Delta \bmod p$. Second, let $3 \nmid \Delta$. Then the polynomial G_Δ from Theorem 2.2.18 is a class polynomial. Again, let p be a prime as in the theorem of Deuring. Then the splitting behavior of $G_\Delta \bmod p$ and $H_\Delta \bmod p$ is the same. In addition, if g_p denotes a root of $G_\Delta \bmod p$, then g_p^3 is a root of $H_\Delta \bmod p$.

2.3.5 Elliptic Curves in Cryptography

In this section we discuss elliptic curves in cryptography. The main purpose of our discussion is to present the requirements for a group of rational points of an elliptic curve to resist all known attacks. As our algorithm shall generate elliptic curves being in conformance with the German Digital Signature Act, we present the conditions published by the German Information Security Agency (GISA). However, GISA restricts to finite prime fields and finite fields of characteristic 2. In the latter case point counting is superior to our approach. Hence we only consider elliptic curves over finite prime fields of large characteristic.

Let $\mathbb{F}_p$ be a finite prime field of large characteristic, say for instance $p \geq 2^{159}$. In addition, let E be an elliptic curve defined over $\mathbb{F}_p$. Once we know that $E(\mathbb{F}_p)$ is an Abelian group, we can define the elliptic curve discrete logarithm problem as usual.

Definition 2.3.25 *Let E be an elliptic curve over a finite prime field $\mathbb{F}_p$. In addition, let $G \in E(\mathbb{F}_p)$ be a point of order N. For an integer l, $1 \leq l \leq N - 1$ set $P = [l]G$. Given $\mathbb{F}_p$, E, G, and P, the* elliptic curve discrete logarithm problem *is to determine the integer l.*

We abbreviate the elliptic curve discrete logarithm problem as ECDLP. In order for an ECDLP to be intractable, that is the integer a may not be computed in reasonable time in practice, we have to choose the parameters such that we can preclude the success of all known algorithms to solve the ECDLP. A detailed discussion of all currently known attacks may be found in [BSS99], Chapter V, [Eng99], Chapter 4, or [Men93], Chapter 5. We call the elliptic curve E *cryptographically strong* if it satisfies the following conditions which make the cryptosystems, in which E is used, secure and efficient.

We first consider security. In order to make the application of known discrete logarithm algorithms impossible, we require that E satisfies the following conditions ([GIS01], section 3.3a, p. 5-6). We remark that these conditions are very strong, as we will see below.

1. We have $|E(\mathbb{F}_p)| = r \cdot k$ with a prime $r > 2^{159}$ and a positive integer $k \leq 4$.

2. The primes r and p are different.

3. The order of p in $\mathbb{F}_r^\times$ is at least 10^4.

4. The class number of the maximal order which contains $\mathrm{End}(E)$ is at least 200.

The condition that r is at least a 160-bit prime excludes the application of ECDLP algorithms whose running time is roughly the square root of the largest prime factor of the group order (see for example [vOM99]). Hence r is often referred to as the *cryptographic prime factor*. In addition, the positive integer k is called the *cofactor*. The condition $k \leq 4$ is for efficiency reasons and will be discussed below. The second condition makes the anomalous curve attack impossible (see for instance [Sem98], [Sma99]). The third condition excludes attacks which reduce the discrete logarithm problem in $E(\mathbb{F}_p)$ to the discrete logarithm problem in a finite extension field of $\mathbb{F}_p$. The degree of this extension over $\mathbb{F}_p$ is at least the order of p in $\mathbb{F}_r^\times$. The reductions make use of the Weil-pairing and the Tate-pairing, respectively. They are due to Menezes, Okamoto, Vanstone ([MOV91]) and Frey, Rück ([FR94],[FMR98]), respectively. The third condition is based on the assumption that the discrete logarithm problem (DLP) in a finite field, whose degree is at least 10^4 over $\mathbb{F}_p$ is intractable. For instance, if $p \approx 2^{160}$, we would have to solve the discrete logarithm problem in a finite field of size about $2^{1600000}$, which is a rather strong requirement. According to widely accepted international standards such as [X9.62], [P1363], or [SEC1] the DLP in a finite field of cardinality about 2^{2000} is considered not to be computable in practice. For instance, [SEC1] requires the order of p in $\mathbb{F}_r^\times$ to be at least 20 (see [SEC1], chapter 3.1.1.1, p. 17). It is well known that for supersingular curves the order of p in $\mathbb{F}_r^\times$ is at most 6. Hence the third condition implies that the endomorphism ring $\mathrm{End}(E)$ is an imaginary quadratic order, i.e. E is ordinary.

Finally, the fourth condition is an additional and exclusive security requirement of the GISA; it does not find consensus in the cryptographic community and may not be found in any standard of elliptic curve cryptography. In addition, it is not clear why the class number of the maximal order containing $\mathrm{End}(E)$ defines the security level, and not the class number of $\mathrm{End}(E)$ itself. We quote a leading cryptograph in the field of elliptic curve cryptography in view of this discussion (the quotation is part of some comments regarding our paper [Bai01c]). "Some researchers (presumably including the authors, in view of their endorsement of requirement 4 at the top of p. 4) have been telling the German Information Security Agency that they must insist on curves whose CM ring has class number at least 200. There is no known reason for wanting this. In particular, it forces one to exclude the curves defined over $\mathbb{F}_2$ that have been shown by Solinas to be very efficient. The only justification I have seen for the ridiculous requirement 4 is that some day someone might find a way to use smallness of the class group in an attack. But no one has ever suggested a way this could be done."

However, we discuss the supporters' argument of this requirement. Let h denote the class number of $\mathrm{End}(E)$. Among all ordinary elliptic curves over $\mathbb{F}_p$ only very few have endomorphism rings with small class numbers, say $h \leq 50$. So those curves may be subject to specific future attacks. However, no such attacks are known. Nevertheless, we point out that in order to generate curves being in conformance with the German Digital Signature Act we have to respect condition 4.

As we will see in Chapter 8 the running time of the generating algorithm in practice highly depends on the class number. Hence it was believed in the cryptographic community that elliptic curves with $h \geq 200$ are not computable using complex multiplication in reasonable time. It is one of the aims of this thesis to prove that the contrary is true.

Let us now consider efficiency. Suppose that an elliptic curve E over a prime field $\mathbb{F}_p$ satisfies the security conditions. If this curve is used in a cryptosystem, the efficiency of this system depends on the efficiency of the arithmetic in $\mathbb{F}_p$. So p should be as small as possible. It follows from the theorem of Hasse (Theorem 2.3.18) that

$$\left(\sqrt{|E(\mathbb{F}_p)|} - 1\right)^2 \leq p \leq \left(\sqrt{|E(\mathbb{F}_p)|} + 1\right)^2 . \tag{2.28}$$

Hence, we try to make $|E(\mathbb{F}_p)|$ as small as possible. Now the first security condition implies $|E(\mathbb{F}_p)| = r \cdot k$ with a prime number $r > 2^{159}$ and the cofactor k. The security of the cryptosystem, in which E is used, is based on the intractability of the discrete logarithm problem in the subgroup of order r in $E(\mathbb{F}_p)$. This security is independent of k. Therefore, k can be as small as possible. In our algorithm $k = 1$ is not always possible. However, $k \leq 4$ is a suitable choice, as we will prove in Section 4.3.

Part II

The Generating Algorithm

Chapter 3

Overview

We present our main algorithm `cryptoCurve`(r_0, k_0, h_0), which bases on the theory presented in Chapter 2. The algorithm consists of various, highly optimized subalgorithms, which we develop in the following sections. We will show that `cryptoCurve` is very fast in practice even for large class numbers. Its input are positive integers r_0, k_0, and h_0. `cryptoCurve` returns a fundamental imaginary quadratic discriminant Δ with $h(\Delta) \geq h_0$. In addition, it returns rational primes p and r, and a positive integer k with $r \geq r_0$, $k \leq k_0$, and $\lfloor \log_2 p \rfloor = \lfloor \log_2 rk \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$; the last property ensures that the prime p is of minimal bitlength for given r_0 and k_0. Finally, `cryptoCurve` returns a cryptographically strong elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$, and a point $G \in E(\mathbb{F}_p)$ of order r. In order to be in conformance with the requirements of GISA (see Section 2.3.5), we have to ensure $r_0 \geq 2^{159}$, $k_0 \leq 4$, and $h_0 \geq 200$. We introduce two terms, which we need in our further discussion.

Definition 3.0.1 **1.** *Let Δ be an imaginary quadratic discriminant. A rational prime p is* suitable for the CM-method of discriminant Δ, *if there exists a $\pi \in \mathcal{O}_\Delta$ with $p = \pi\overline{\pi}$.*
2. *Let $I = [i_0, i_1] \subset \mathbb{N}$ and let $K \in \mathbb{N}$ with $mK = i_1 - i_0 + 1$. For $0 \leq s \leq K-1$ let I_s denote the s-th subinterval of I of length m, that is $I_s = [i_0 + sm, i_0 + (s+1)m - 1]$. A subset $\mathcal{S} \subset I$ is* K-uniformly distributed in I *if $|\mathcal{S} \cap I_{s_1}| = |\mathcal{S} \cap I_{s_2}|$ for all $0 \leq s_1 < s_2 \leq K-1$.*

We first discuss some requirements on the prime p. First, p has to be suitable for the CM-method of discriminant Δ. Second, for efficiency reasons, the prime p should be as small as possible. Obviously, due to the theorem of Hasse, if r_0 and k_0 are given, $\lfloor \log_2 r_0 k_0 \rfloor + 1$ is the maximal lower bound of the bitlength of p. Hence, if we set $b = \lfloor \log_2 r_0 k_0 \rfloor$, $i_0 = 2^b$, and $i_1 = 2^{b+1} - 1$, the algorithm will output primes $p \in [i_0, i_1]$. Finally, for security reasons, the primes p should not be special in any sense. More precisely, p should be uniformly distributed in the set of primes in $[i_0, i_1]$, which are suitable for the CM-method of discriminant Δ. However, in Section 4.2 we will see, that this requirement is too strong for our efficient algorithms. Hence, we weaken this requirement by considering K-uniform distributions. To summarize, the primes p returned by `cryptoCurve`(r_0, k_0, h_0) have the following properties:

(P1) p is suitable for the CM-method of discriminant Δ.

(P2) $p \in [i_0, i_1]$, where $b = \lfloor \log_2 r_0 k_0 \rfloor$, $i_0 = 2^b$, and $i_1 = 2^{b+1} - 1$.

(P3) The distribution of the primes is K-uniformly in $[i_0, i_1]$ for a large K, say $K \geq 2^{40}$.

`cryptoCurve` first checks whether the input fits to the requirements of Section 2.3.5; if this is not the case, the algorithm outputs an error message and terminates. Otherwise, `cryptoCurve` invokes two subalgorithms, `findPrime` and `findCurve`, which we discuss in what follows.

Algorithm 3.1: `cryptoCurve`(r_0, k_0, h_0)

Input: Positive integers r_0, k_0, and h_0.
Output: A fundamental imaginary quadratic discriminant Δ with $h(\Delta) \geq h_0$.
Rational primes p and r, and a positive integer k with $r \geq r_0$, $k \leq k_0$, and $\lfloor \log_2 p \rfloor = \lfloor \log_2 rk \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$.
A cryptographically strong elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
A point $G \in E(\mathbb{F}_p)$ of order r.

if $r_0 < 2^{159}$ OR $k_0 > 4$ OR $h_0 < 200$ **then**
  Output("Invalid input."); terminate;
$(\Delta, p, r, k) \leftarrow$ `findPrime`(r_0, k_0, h_0);
$(E, G) \leftarrow$ `findCurve`(Δ, p, r, k);
return (Δ, p, r, k, E, G);

We explain algorithm `findPrime`. Its task is as follows: Let r_0, k_0, and h_0 be given. We have to find a fundamental imaginary quadratic discriminant Δ of class number at least h_0. Furthermore, we have to determine a rational prime p, which may be written as $4p = t^2 - \Delta y^2$ with $(t, y) \in \mathbb{Z}^2$, such that either $p + 1 - t = rk$ or $p + 1 + t = rk$ holds with a prime $r \geq r_0$ and a positive integer $k \leq k_0$. In addition, rk has to be the cardinality of a group of rational points of a cryptographically strong elliptic curve over $\mathbb{F}_p$.

We investigate two different approaches of `findPrime`. It turns out that the first one is much faster in practice.

1. Our very efficient first approach first fixes a fundamental discriminant Δ with $h(\Delta) \geq h_0$. Once, Δ is chosen, we determine $p = \frac{t^2 - \Delta y^2}{4}$, r, and k. As Δ is fixed, we call this proceeding *Fixed Discriminant Approach*. The corresponding algorithm `findPrimeDeltaFixed` is discussed in Chapter 4. We compare different algorithms and give upper bounds for their complexities. For instance, we prove that the complexity of `findPrimeDeltaFixed` is at most $O(h^2 \log^5(r_0 k_0)/|\Delta|)$. We derive necessary conditions on t and y for $\frac{t^2 - \Delta y^2}{4}$ to be prime and either $p + 1 - t$ or $p + 1 + t$ to be not divisible by small, odd primes (i.e. 3,5,7). In addition, we introduce sieving methods. It turns out that this approach is the most efficient one in practice. For example, a sample input of $r_0 = 2^{159}$, $k_0 = 4$, and $h_0 = 200$ gives a running time of about 0.2 seconds on the Pentium III. Finally, we investigate the conformance of our theoretical complexity considerations with practical results. We are not aware of any comparably efficient algorithm.

2. Our second design of `findPrime` first fixes a rational prime p and finds a suitable discriminant Δ afterwards. As the field is chosen first, we name this approach *Fixed Field Approach*. We present the according algorithm `findDiscriminant` in Chapter 5. Although this approach was already proposed by Atkin, Morain [AM93], [X9.62], and [P1363], we show how to optimize it using Legendre symbols and pre-computations. The idea of using Legendre symbols was already described by Atkin and Morain. However, the applicability of these ideas to cryptography and the implementation of the optimization is a new contribution.

We next give a formal description of `findPrime`. Its input are the bounds r_0, k_0, and h_0. We first decide which approach we follow. If we set $p = 0$, we use the Fixed Discriminant Approach. If $p \neq 0$, we assume p to be prime and follow the Fixed Field Approach. Both subalgorithms and hence `findPrime` return a fundamental discriminant Δ and appropriate p, r, and k.

In order to decide whether a given positive integer n is prime, we make use of the function `isPrime`(n, m). `isPrime` requires two positive integers n and m as input. It implements the Miller-Rabin probabilistic primality test as described in [BS96], p.282. The positive integer m is the number of independent Miller-Rabin tests performed on n. `isPrime` returns `true` if n cannot be shown to be composite by m independent Miller-Rabin tests. Otherwise, `isPrime` returns `false`. According to common standards (e.g. [X9.62], A.2.1, p. 46) $m = 50$ is sufficient in practice.

Algorithm 3.2: `findPrime`(r_0, k_0, h_0)

Input: Positive integers r_0, k_0, and h_0.
Output: A fundamental imaginary quadratic discriminant Δ with $h(\Delta) \geq h_0$.
Rational primes p and r, and a positive integer k with $r \geq r_0$, $k \leq k_0$, and $\lfloor \log_2 p \rfloor = \lfloor \log_2 rk \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$, such that a cryptographically strong elliptic curve E over $\mathbb{F}_p$ exists with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.

Input p; $m \leftarrow 50$; //set the number of Miller-Rabin tests to 50
if $p = 0$ **then**
  `findPrimeDeltaFixed`(r_0, k_0, h_0);
else if `isPrime`$(p, m) =$ `true` AND $\lfloor \log_2 p \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$ **then**
  `findDiscriminant`(p, r_0, k_0, h_0);
else
  Output("Invalid p"); terminate;

Once, Δ, p, r, and k are known, `cryptoCurve` invokes `findCurve`(Δ, p, r, k). Let H_Δ be the Hilbert polynomial corresponding to $\mathcal{O}_\Delta$. We denote it by H in what follows. From Section 2.3.4 we know that H splits modulo p into pairwise different linear factors. Let j_p be a zero of H mod p, s_p a quadratic non-residue mod p, and $(a_p, b_p) = (3\kappa_p, 2\kappa_p)$ with $\kappa_p = j_p/(1728 - j_p)$. Then one of the elliptic curves (a_p, b_p) or $(a_p s_p^2, b_p s_p^3)$ is of order rk. By choosing points on each curve and testing whether their order is a divisor of rk, the curve of order rk can easily be identified.

It remains to explain algorithm `findRoot`(Δ, p). Given the imaginary quadratic discriminant Δ and the prime p, `findRoot` returns a root j_p of the Hilbert polynomial H modulo p. First, we have to determine the $h(\Delta)$ reduced representatives of $C(\Delta)$. We will address this problem in Chapter 6: Our algorithm `classGroup`(Δ) requires the discriminant Δ and returns an array R storing the $h(\Delta)$ reduced representatives of discriminant Δ. In addition, `classGroup` returns the class number $h(\Delta)$.

Next, we have to compute a generating polynomial of the Hilbert class field. As stated in Section 2.2.3 depending on Δ mod 24, we may use alternative polynomials with smaller coefficients than the Hilbert polynomial H. According to Definition 2.2.14 and Theorems 2.2.18 and 2.2.19 we have to evaluate the η-function, and either the Weber functions $\mathfrak{f}$, $\mathfrak{f}_1$, and

Algorithm 3.3: `findCurve`(Δ, p, r, k)

Input: A fundamental imaginary quadratic discriminant Δ.
Rational primes p and r and a positive integer k such that there exists an elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
Output: An elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
A point $G \in E(\mathbb{F}_p)$ of order r.

$j_p \leftarrow$ `findRoot`(Δ, p);
Select a quadratic non-residue s_p mod p; $\kappa_p \leftarrow j_p/(1728 - j_p)$; $a_p \leftarrow 3\kappa_p$; $b_p \leftarrow 2\kappa_p$;
$E_1 \leftarrow (a_p, b_p)$; $E_2 \leftarrow (a_p s_p^2, b_p s_p^3)$;
while `true` **do**
  $Q_1 \in_R E_1(\mathbb{F}_p) \setminus \{O\}$; $Q_2 \in_R E_2(\mathbb{F}_p) \setminus \{O\}$; //Choose random, non-trivial points
  if $kQ_1 \neq O$ AND $krQ_1 = O$ **then**
    return (E_1, kQ_1);
  else if $kQ_2 \neq O$ AND $krQ_2 = O$ **then**
    return (E_2, kQ_2);

$\mathfrak{f}_2$, the function γ_2, or the modular function j, respectively. Hence, in Chapter 7 we investigate in detail efficient algorithms for computing these functions. The result of Section 7 is that using a representation of the η-function as a sum is the most efficient way to evaluate $\eta(\tau)$ in practice. In addition, we will show that it is optimal to compute the further functions by a representation via the η-function. Our algorithm `computeClassInvariants`(Δ, h, R) implements our optimized algorithms of Chapter 7. It requires the discriminant Δ, its class number $h(\Delta)$, and an array R storing the $h(\Delta)$ reduced representatives of discriminant Δ as input. Depending on the value of Δ mod 24, the algorithm chooses a generating polynomial of the Hilbert class field with rather small coefficients, as explained in Section 2.2. We denote such a class polynomial by C. Using the most efficient representation of the according numbertheoretical function, `computeClassInvariants` computes the $h(\Delta)$ roots of the chosen class polynomial C; it returns them as an array I.

Algorithm 3.4: `findRoot`(Δ, p)

Input: A fundamental imaginary quadratic discriminant Δ and a rational prime p which is suitable for the CM-method of discriminant Δ.
Output: A root j_p of the Hilbert polynomial of discriminant Δ modulo p.

$(R, h) \leftarrow$ `classGroup`(Δ);
$I \leftarrow$ `computeClassInvariants`(Δ, h, R);
$C \leftarrow$ `computeClassPolynomial`(I, h);
$c_p \leftarrow$ `find_root`(p, C);
if $\Delta \equiv 1 \bmod 8$ AND $3 \nmid \Delta$ **then**
  return $((c_p^{24} - 16)^3 / c_p^{24})$;
else if $3 \nmid \Delta$ **then**
  return (c_p^3);
else
  return (c_p);

Once, all the roots of C are computed, we have to compute C itself. In Chapter 8 we present our optimized algorithm for this task. It turns out that generalizing an idea due to Karatsuba ([Kar95]) yields a very efficient algorithm. Our algorithm implementing these ideas is named `computeClassPolynomial`(I). As input the algorithm gets an array I holding the roots of C; `computeClassPolynomial` returns the according class polynomial C.

Finally, we have to determine a root of C mod p and map it to a root of H mod p. We denote a root of C mod p and H mod p by c_p and j_p, respectively. In order to compute c_p we make use of the LiDIA-function `find_root`(p, C). As input this function requires a prime p and a polynomial $C \in \mathbb{Z}[X]$ which splits into linear factors modulo p. It returns a zero of C mod p. `find_root` uses the Cantor-Zassenhaus split as explained in [Coh95], section 3.4.4, p.126-127, and a very efficient polynomial arithmetic due to Shoup ([Sho95]). As explained on Page 33, depending on the value of Δ mod 24, `findRoot` maps c_p to j_p and returns j_p.

In addition, we investigate the floating point precision for computing a class polynomial in Chapter 9. We present explicit formulae. In case of the Weber polynomial W, basing on a large database of practical tests, we develop a new formula yielding a precision, which is significantly better than any previously proposed one. Finally, in Section 10.1 we investigate the complexity of algorithm `cryptoCurve`.

We conclude this section with our algorithm `isStrong`(r_0, k_0, p, N). It gets the bounds r_0 and k_0 as input. In addition, it requires a prime p and the order N of an elliptic curve over $\mathbb{F}_p$. `isStrong` returns the prime r, if $N = rk$ is the order of a cryptographically strong elliptic curve over $\mathbb{F}_p$ with $r \geq r_0$ and $k \leq k_0$, respectively. Otherwise `isStrong` returns 0. In order to decide whether N is the order of a cryptographically strong elliptic curve over $\mathbb{F}_p$ `isStrong` implements the requirements of Section 2.3.5. As we ensure $h(\Delta) \geq h_0$ in our algorithm `findPrime`, we do not have to check this condition in `isStrong`.

Algorithm 3.5: `isStrong`(r_0, k_0, p, N)

Input: Positive integers r_0 and k_0.
A prime p and the order N of a group of rational points of an elliptic curve over $\mathbb{F}_p$.
Output: A prime r if $N = rk$ is the order of a cryptographically strong elliptic curve over $\mathbb{F}_p$ with $r \geq r_0$ and $k \leq k_0$, and 0 otherwise.

$m \leftarrow 50$; //set the number of Miller-Rabin tests to 50
$r \leftarrow 0$; $k \leftarrow 0$; //initialize r and k with 0, respectively
//check by trialdivision if cofactor is at most k_0; if not, return 0
for $i \leftarrow 1; i \leq k_0; i \leftarrow i + 1$ **do**
  if $i \mid N$ AND `isPrime`$(N/i, m) =$ `true` **then**
    $r \leftarrow N/i$; $k \leftarrow i$; break;
if $r = 0$ **then**
  return (0);
//check if curve is anomalous; if yes, return 0
if $p = r$ **then**
  return (0);
//check if order of p in $\mathbb{F}_r^\times$ is at least 10^4
$p_r \leftarrow 1 \bmod r$;
for $i \leftarrow 1; i \leq 10^4; i \leftarrow i + 1$ **do**
  $p_r \leftarrow p \cdot p_r \bmod r$;
  if $p_r = 1$ **then**
    return (0);
return (r);

Chapter 4

Finding a Suitable Cardinality: A Fixed Discriminant Approach

In this chapter we develop our very efficient algorithm `findPrimeDeltaFixed`(r_0, k_0, h_0). Given r_0, k_0 and h_0, the algorithm first chooses a fundamental discriminant Δ of class number at least h_0 as explained below. Once, Δ is chosen, `findPrimeDeltaFixed` computes a prime p suitable for the CM-method of discriminant Δ such that there exists a cryptographically strong elliptic curve over $\mathbb{F}_p$ with endomorphism ring of discriminant Δ. The algorithm returns Δ, p, r, and k. `findPrimeDeltaFixed` will be presented in Section 4.3.

We investigate in detail algorithms to find primes suitable for the CM-method of discriminant Δ, if the discriminant Δ is fixed. In this chapter, we discuss two different approaches: Our first one, which we present in Section 4.1, first chooses a prime p and decides whether it is suitable for the CM-method of discriminant Δ. The second one, presented in Section 4.2, starts with a pair $(t, y) \in \mathbb{Z}^2$ and tests $\frac{t^2-\Delta y^2}{4}$ for primality. As we randomly choose the trace t we name this algorithm `generatePrimeRandomTrace`. It turns out that the latter approach is faster in both theory and practice.

Then, in Section 4.3 we turn to the problem of finding appropriate values p, r, and k for given r_0, k_0, and h_0. As the approach of finding primes suitable for the CM-method of Section 4.1 is rather slow, we only take algorithm `generatePrimeRandomTrace` into account. We first extend it to an algorithm named `generateTwinPrimeRandomTrace`. This algorithm is non-optimized. However, we prove that its bit-complexity is $O(h^2 \log^5(r_0 k_0)/|\Delta|)$. Hence, we deduce that the bit-complexity of our optimized algorithm `findPrimeDeltaFixed` is bounded by $O(h^2 \log^5(r_0 k_0)/|\Delta|)$. In addition, we give evidence that `findPrimeDeltaFixed` is very efficient in practice. For instance, if $r_0 = 2^{160}$ and $k_0 \leq 4$, its running time is about 0.2 seconds on the Pentium III for discriminants of class number 200.

In order to implement `findPrimeDeltaFixed`, we make use of our database delta_h. delta_h stores all discriminants Δ with $|\Delta| < 10^8$ according to their class numbers. If the user does not specify a discriminant, using the database delta_h, `findPrimeDeltaFixed` chooses the largest fundamental discriminant Δ of class number h_0. For instance, $\Delta = -21311$ is the largest fundamental discriminant of class number 200. Hence, we often make use of this discriminant in our practical tests.

In our investigation we will search for primes p in an interval $I = [i_0, i_1]$. Mostly, i_0 and i_1 will

be of the form $i_0 = 2^b$ and $i_1 = 2^{b+1} - 1$ for some positive integer b, respectively. The reason is that only the bitlength of the prime r is of cryptographic relevance, where we assume rk to be the order of a cryptographically strong elliptic curve over $\mathbb{F}_p$. Hence, in practice, r_0 is of the form 2^{b_0}. In addition, $k_0 \in \{1; 2; 4\}$. According to the theorem of Hasse (Theorem 2.3.18) we mostly have $\lfloor \log_2 r_0 k_0 \rfloor = \lfloor \log_2 rk \rfloor = \lfloor \log_2 p \rfloor$. Thus we search for a prime p with $p \in [2^{b_0} k_0, 2^{b_0+1} k_0 - 1]$. Typical values are $b_0 \in \{159; 160\}$ and $k_0 \in [1; 4]$. Hence we often set $i_0 = 2^b$ with $b \in \{159; 160; 161; 162\}$.

4.1 Finding Primes Suitable for the CM-method: Choosing the Prime first

In this section we discuss two approaches for the following problem: Let an interval $I = [i_0, i_1]$ and an imaginary quadratic discriminant Δ be given. Find a prime $p \in I$ suitable for the CM-method of discriminant Δ. It turns out that the primes returned by both approaches are uniformly distributed in the set of primes in I which are suitable for the CM-method of discriminant Δ.

Our first approach discussed in Section 4.1.1 simply chooses random integers in I until a prime suitable for the CM-method of discriminant Δ is found. The second algorithm presented in Section 4.1.2 chooses random primes in I until a suitable one is found. It turns out that the bit-complexity of both algorithms is $O(h \cdot \log^4 i_1)$. However, both algorithms are rather slow in practice.

4.1.1 Finding Suitable Primes by Randomly Choosing Integers

In this section we present our first approach to find a prime p in a given interval I which is suitable for the CM-method of a given discriminant Δ. We name this algorithm `generateIsPrime`. It is rather trivial: Given Δ and I, we randomly choose an integer $n \in I$ and test it for primality; again, we make use of the function `isPrime`(n, m) as explained on Page 41. If n turns out to be prime, we have to check whether it is suitable for the CM-method of discriminant Δ. In order to do so, we make use of an algorithm due to Cornacchia ([Coh95], Section 1.5.2, p. 34-36): `cornacchia`(Δ, p) requires an imaginary quadratic discriminant Δ and an odd prime p with $4p > |\Delta|$ as input. It returns 0, if the equation $4p = t^2 - \Delta y^2$ does not have a solution $(t, y) \in \mathbb{Z}^2$, and t of an integer solution (t, y) with $t > 0$ otherwise.

`generateIsPrime` obviously has the properties (P1) and (P2) of Chapter 3. In addition, as the integers are chosen at random, `generateIsPrime` returns primes being uniformly distributed in the set of primes, which are suitable for the CM-method of discriminant Δ. This property is stronger than property (P3) of Chapter 3. Furthermore, our practical results indicate that the primes suitable for the CM-method of discriminant Δ are not uniformly distributed in a given interval I. For example, using `generateIsPrime`$(-21311, [2^{162}, 2^{163} - 1])$, we generated 20000 primes. The distribution of the primes is plotted in Figure 4.1.

We next investigate the bit-complexity of `generateIsPrime`. First, we estimate the probability P that `generateIsPrime` succeeds for a randomly chosen n. We need the following term (a more general definition may be found in [Cox89], p. 169):

Algorithm 4.1: `generateIsPrime`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$.
An interval $[i_0, i_1] \subset \mathbb{N}$.
Output: A prime $p \in [i_0, i_1]$ and an integer t such that $4p = t^2 - \Delta y^2$ with an integer y.

```
m ← 50; //set the number of Miller-Rabin tests to 50
while true do
  n ∈_R [i_0, i_1];
  if isPrime(n, m) = true then
    t ← cornacchia(Δ, n);
    if t ≠ 0 then
      return (n, t);
```

Definition 4.1.1 *Let $\mathbb{P}$ denote the set of rational primes, and let $\mathcal{S}$ be a subset of $\mathbb{P}$. The **Dirichlet density** $\delta(\mathcal{S})$ of $\mathcal{S}$ is defined as*

$$\delta(\mathcal{S}) = \lim_{s \to 1^+} \frac{\sum_{p \in \mathcal{S}} p^{-s}}{-\log(s-1)}$$

provided the limit exists.

We remark that the Dirichlet density $\delta(\mathcal{S})$ actually represents the density of $\mathcal{S}$ in $\mathbb{P}$. The following theorem provides the Dirichlet density of the primes suitable for the CM-method of discriminant Δ. We adapt it from a more general theorem in [Cox89] (Theorem 9.12, p. 188); we remark that in Theorem 9.12 of [Cox89] the densities $\frac{1}{h(\Delta)}$ and $\frac{1}{2h(\Delta)}$ are given in the wrong order.

Theorem 4.1.2 *Let $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$. Set $\mathcal{S} = \{p \in \mathbb{N} : p \text{ prime}, p = \pi \cdot \overline{\pi} \text{ for a } \pi \in \mathcal{O}_\Delta\}$. Then the Dirichlet density $\delta(\mathcal{S})$ exists and is equal to $\frac{1}{2h}$, where h denotes the class number of $\mathcal{O}_\Delta$.*

Let $\mathcal{S}$ be as in Theorem 4.1.2. We make the following assumption: If I is not "too small" (e.g. $i_1 - i_0 + 1 \geq 2^{159}$) the asymptotic formula $\delta(\mathcal{S}) = \frac{1}{2h}$ is valid in I. Making use of this assumption and the logarithmic integral li, our approximation of P is

$$P = \delta(\mathcal{S}) \cdot \frac{\sharp\{p \in I,\ p \text{ prime}\}}{\sharp\{x \in I\}} = \frac{1}{2h} \cdot \frac{\mathrm{li}(i_1) - \mathrm{li}(i_0)}{i_1 - i_0 + 1}\ . \tag{4.1}$$

We remark that we estimate both the number of primes in I by computing $\mathrm{li}(i_1) - \mathrm{li}(i_0)$ and the proportion of primes which are suitable for the CM-method by the Dirichlet density $\delta(\mathcal{S})$. However, our practical tests indicate that $\mathrm{li}(i_1) - \mathrm{li}(i_0)$ approximates well the number of primes in I. Furthermore, the results of our practical experiments in Table 4.1 affirm Formula (4.1).

Example: For cryptographic purposes we have the boundary condition $h_0 \geq 200$. Furthermore, assume $I = [2^{162}, 2^{163} - 1]$. Formula (4.1) yields

$$P = \frac{1}{400} \cdot \frac{\mathrm{li}(2^{163} - 1) - \mathrm{li}(2^{162})}{2^{162}} = \frac{1}{400} \cdot \frac{5.1883 \cdot 10^{47}}{2^{162}} \approx \frac{1}{45070}\ .$$

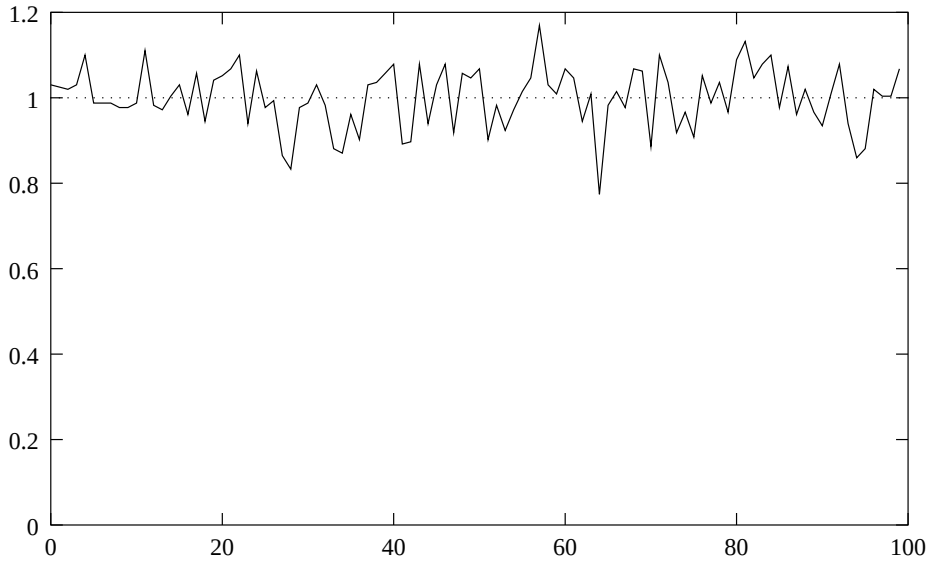


Figure 4.1: Distribution of primes returned by `generateIsPrime`$(-21311, [2^{162}, 2^{163}-1])$. We evaluate 20000 primes. We subdivide the interval $[2^{162}, 2^{163}-1]$ in 100 subintervals charted at the abscissae. In y-direction we illustrate the ratio of the number of primes in a subinterval to the mean value 200.

Hence, before succeeding we expect to pass approximately 45000 times through the while-loop of `generateIsPrime`. This means that on average we have to choose 45000 random integers in I and test all these integers for primality. This is rather burdensome. Practical results can be found in Table 4.1. These results affirm our theoretical estimation.

Δ	Number of tests	Number of tested integers	Number of tested primes	Time of `generateIsPrime` in seconds
-21311	5000	44782	395.67	179.05
-23639	5000	43906	388.59	168.28
-24311	5000	46201	409.84	178.76
-25151	5000	45071	400.19	174.25

Table 4.1: Data delivered by `generateIsPrime`$(\Delta, [2^{162}, 2^{163}-1])$ for some fundamental discriminants of class number 200. The running time was measured on the SUN UltraSPARC I.

We are now able to determine the bit-complexity of `generateIsPrime`. In practice we will only make use of $i_1 = 2i_0 - 1$. Hence we restrict to this case.

Theorem 4.1.3 *Let i_0 be a positive integer and Δ an imaginary quadratic discriminant of class number h. If $i_1 = 2i_0 - 1$, the bit-complexity of* `generateIsPrime`$(\Delta, [i_0, i_1])$ *is* $O(h \cdot \log^4 i_1)$.

Proof: For each tested integer n we have to perform a Miller-Rabin primality test of bit-complexity $O(\log^3 n)$ ([Coh95], Section 8.2, p. 415). According to Formula (4.1) we expect to

test $2h \cdot \frac{i_1 - i_0 + 1}{\mathrm{li}(i_1) - \mathrm{li}(i_0)} \sim 2h \cdot \frac{i_1/2}{\pi(i_1) - \pi(i_1/2)}$ integers. We have

$$\begin{aligned} \pi(i_1) - \pi(i_1/2) &\sim \frac{i_1}{\log i_1} - \frac{i_1/2}{\log i_1/2} \\ &\sim \frac{i_1}{\log i_1}\left(1 - \frac{1}{2} \cdot \frac{1}{1 - \frac{\log 2}{\log i_1}}\right) \\ &\sim \frac{1}{2} \cdot \frac{i_1}{\log i_1} \, . \end{aligned}$$

Thus the bit-complexity of all Miller-Rabin tests is $2h \cdot \frac{i_1/2}{i_1/(2 \log i_1)} \cdot \log^3 i_1 = O(h \log^4 i_1)$.
If a tested integer n turns out to be a probable prime, we have to perform the algorithm of Cornacchia. Its complexity is essentially the running time of computing a square root modulo n and performing the Euclidian algorithm on integers of order of magnitude of n (see [Coh95], Section 1.5.2, p. 34-36). We make use of Shank's RESSOL for computing square roots. Its complexity is $O(\log^4 n)$ ([Coh95], Section 1.5.1, p. 33); in addition, the bit-complexity of the Euclidian step is at most $O(\log^3 n)$ ([Coh95], Section 1.3, p. 13). Thus the running time of Cornacchia's algorithm is $O(\log^4 n)$. Theorem 4.1.2 states that we expect to pass $2h$ times through Cornacchia's algorithm. Thus the complexity of this part is $O(h \cdot \log^4 i_1)$.
We conclude that the bit-complexity of `generateIsPrime` is $O(h \cdot \log^4 i_1)$. □

4.1.2 Finding Suitable Primes by Randomly Choosing Primes

Our next approach extends algorithm `generateIsPrime`; we call it `generateNextPrime`. Its underlying idea is as follows: As in the previous section we assume that an imaginary quadratic discriminant Δ and an interval $I = [i_0, i_1]$ are given. However, in order to avoid choosing integers independently in a given range, we take a random integer n in the interval I and search for the next prime $p \in I$, that is we set $p = \min\{x \in I : x \geq n, x \text{ prime}\}$, if such a prime p exists. For given positive integers n and m, the function `nextPrime`(n, m) returns $\min\{x \in \mathbb{Z} : x \geq n, \texttt{isPrime}(x, m) = \texttt{true}\}$. Once, p is found, we have to decide whether p is suitable for the CM-method of discriminant Δ. Again we make use of the function `cornacchia`(Δ, p) introduced on Page 46.

As in the previous section we assume that the asymptotic formula $\delta(\mathcal{S}) = \frac{1}{2h}$ is valid in I. The main advantage of `generateNextPrime` with respect to `generateIsPrime` is that we expect to choose only $2h$ random integers n before `generateNextPrime` terminates. For instance, if Δ has class number 200, we expect to choose 400 random integers n to find a prime p suitable for the CM-method of discriminant Δ. Figure 4.2 plots the distribution of primes returned by `generateNextPrime`$(-21311, [2^{162}, 2^{163} - 1])$. In addition, Table 4.2 lists further practical results. We deduce that our practical results affirm our assertion $\delta(\mathcal{S}) = \frac{1}{2h}$. Furthermore, we see from Table 4.1 and Table 4.2 that `generateNextPrime` is much faster in practice than `generateIsPrime`. Finally, Figure 4.2 indicates that the primes returned by `generateNextPrime` are uniformly distributed within the set of primes suitable for the CM-method of discriminant Δ. Hence, the primes returned by `generateNextPrime` and `generateIsPrime` have the same properties.

Next, we explain how to efficiently implement `nextPrime`(n, m) by using sieving methods. Let P_i denote the array which stores the first i odd primes. For example, we have $P_1 = \{3\}$,

Algorithm 4.2: `generateNextPrime`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$.
An interval $[i_0, i_1] \subset \mathbb{N}$.
Output: A prime $p \in I$ and an integer t such that $4p = t^2 - \Delta y^2$ with an integer y.

```
m ← 50; //set the number of Miller-Rabin tests to 50
while true do
  n ∈_R I;
  p ← nextPrime(n, m);
  if p ∈ I then
    t ← cornacchia(Δ, p);
    if t ≠ 0 then
      return (p, t);
```

Δ	Number of tests	Number of tested primes	Time of `generateNextPrime` in seconds
-21311	10000	400.46	79.621
-23639	5000	405.88	40.425
-24311	5000	401.90	79.396
-25151	5000	397.64	78.194

Table 4.2: Data delivered by `generateNextPrime` for $I = [2^{162}, 2^{163} - 1]$. All discriminants are fundamental with $h = 200$. The running time was measured on the SUN UltraSPARC I.

$P_2 = \{3, 5\}$, $P_3 = \{3, 5, 7\}$. If the randomly chosen integer $n \in I$ is odd, we set $n' = n$; otherwise we set $n' = n + 1$. We then initialize the array R_i with the remainders of n' modulo the primes stored in P_i, that is $R_i = \{n' \bmod 3, n' \bmod 5, \ldots, n' \bmod p_i\}$, where p_i denotes the i-th odd prime. We perform the function `isPrime`(n', m) on the current value of n' if and only if *all* entries of R_i are non-zero. If n' turns out to be composite, we set $n' \leftarrow n' + 2$ and adapt R_i. Our implementation of `nextPrime` uses $i = 7$. We remark that the sieving methods have been already implemented in the according LiDIA-function `next_prime` of the class `bigint`.

Finally, we prove that the complexity of `generateNextPrime` and `generateIsPrime` is the same.

Theorem 4.1.4 *Let i_0 be a positive integer and Δ an imaginary quadratic discriminant of class number h. If $i_1 = 2i_0 - 1$, the bit-complexity of* `generateNextPrime`$(\Delta, [i_0, i_1])$ *is* $O(h \cdot \log^4 i_1)$.

Proof: We first estimate the complexity of `nextPrime`(n, m). Let n be given. According to the prime number theorem the density of primes in I is $\frac{1}{\log i_1}$. Furthermore, the probability that the current value of n' is divisible by none of the primes in P_i is $\prod_{p \in P_i} \frac{p-1}{p} < 1$. Hence, we expect to perform $\log i_1 \cdot \prod_{p \in P_i} \frac{p-1}{p}$ times the Miller-Rabin test of complexity $O(\log^3 i_1)$ (the complexity of a Miller-Rabin test may be found in [Coh95], Section 8.2, p. 415). We neglect the running time of sieving. Thus, the complexity of `nextPrime`(n, m) is $O(\log^4 i_1)$. Furthermore, we have to perform the algorithm of Cornacchia on p if $p \in I$. However, the probability of $p \notin I$ vanishes in practice. As in the proof of Theorem 4.1.3 the complexity

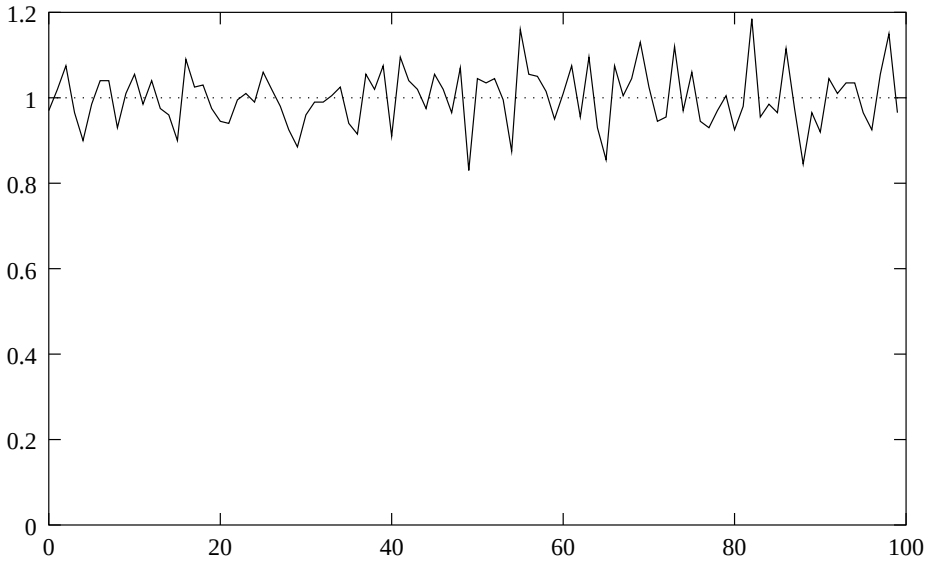


Figure 4.2: Distribution of primes returned by `generateNextPrime`$(-21311, [2^{162}, 2^{163}-1])$. We evaluate 20000 primes. We subdivide the interval $[2^{162}, 2^{163}-1]$ in 100 subintervals charted at the abscissae. In y-direction we illustrate the ratio of the number of primes in a subinterval to the mean value 200.

of `cornacchia`(Δ, p) is $O(\log^4 p) = O(\log^4 i_1)$. As we expect to choose $2h$ initial values n before finding a prime p suitable for the CM-method of discriminant Δ, the bit-complexity of `generateNextPrime` is $O(h \cdot \log^4 i_1)$. □

4.1.3 Comparison of both Approaches

Although `generateIsPrime` and `generateNextPrime` are of same bit-complexity, the latter one turns out to be much faster in practice. We explain this fact. Using `generateIsPrime` we have to perform the Miller-Rabin test for each chosen integer n. However, the fraction of the number of primality tests in algorithm `generateNextPrime` with respect to `generateIsPrime` is $\prod_{p \in P_i} \frac{p-1}{p}$, as we have shown in the proof of Theorem 4.1.4. We set $P_i = \{3, 5, 7, 11, 13, 17, 19\}$ in our implementation. Thus $\prod_{p \in P_i} \frac{p-1}{p} = 0.342048$ follows, and making use of `generateNextPrime` we save about 65% of the timing of the Miller-Rabin test with respect to `generateIsPrime`.

4.2 Finding Primes Suitable for the CM-method: Choosing the Representation first

We discuss another approach for the following problem: Let an interval $I = [i_0, i_1]$ and an imaginary quadratic discriminant Δ be given. Find a prime $p \in I$ suitable for the CM-method of discriminant Δ. From Proposition 2.1.19 we know that a prime p is suitable for the CM-

method of discriminant Δ if and only if p is of the form $p = \frac{t^2-\Delta y^2}{4}$ for some $(t, y) \in \mathbb{Z}^2$. In this section we first choose the representation (t, y) and check whether the integer $\frac{t^2-\Delta y^2}{4}$ is a prime in I; obviously we have to take care of $t \equiv y\Delta \bmod 2$. If p is prime, t is the trace of a prime in $\mathcal{O}_\Delta$ over p.

First, we present our general algorithm `generatePrimeRandomTrace` which implements this idea. We determine its complexity, which is equal to $O(h \cdot \log^4 i_1/\sqrt{|\Delta|})$, where h denotes the class number of Δ. Next, we show that this approach turns out to be much faster in practice than the algorithms of Section 4.1. Then, we introduce the term of a proper representation (t, y); we show how to speed up `generatePrimeRandomTrace` using proper representations. Finally, we discuss whether the primes returned by `generatePrimeRandomTrace` are K-uniformly distributed for some $K \in \mathbb{N}$.

Input of `generatePrimeRandomTrace`$(\Delta, [i_0, i_1])$ is an imaginary quadratic discriminant Δ and an interval I. It returns a prime $p \in I$ and a $t \in \mathbb{N}$ such that $4p = t^2 - \Delta y^2$ with an integer y. Its idea is quite trivial: Randomly choose pairs $(t, y) \in \mathbb{N}^2$ such that $t \equiv y\Delta \bmod 2$ and $n = \frac{t^2-\Delta y^2}{4} \in I$. If n turns out to be prime, the algorithm returns (n, t) and terminates. `generatePrimeRandomTrace` is not optimized. It will be refined in Section 4.3 using congruence conditions and sieving methods.

Algorithm 4.3: `generatePrimeRandomTrace`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 0, 1 \bmod 4$.
An interval $[i_0, i_1] \subset \mathbb{N}$.

Output: A prime $p \in [i_0, i_1]$ and a $t \in \mathbb{N}$ such that $4p = t^2 - \Delta y^2$ with an integer y.

$m \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $t \in_R \{1, \ldots, 2 \cdot \lfloor \sqrt{i_1} \rfloor\}$;
  if $\frac{4i_0-t^2}{|\Delta|} \leq 0$ **then**
    $y \in_R \{1, \ldots, \left\lfloor \sqrt{(4i_1 - t^2)/|\Delta|} \right\rfloor\}$;
  else
    $y \in_R \{\left\lceil \sqrt{(4i_0 - t^2)/|\Delta|} \right\rceil, \ldots, \left\lfloor \sqrt{(4i_1 - t^2)/|\Delta|} \right\rfloor\}$;
  if $t \not\equiv y\Delta \bmod 2$ **then**
    $t \leftarrow t + 1$;
  $n \leftarrow \frac{t^2-\Delta y^2}{4}$;
  if `isPrime`$(n, m) =$ `true` **then**
    return (n, t);

We determine the bit-complexity of `generatePrimeRandomTrace`$(\Delta, [i_0, i_1])$. We make use of a theorem due to Gauss ([Coh62], Theorem 2, p. 160).

Theorem 4.2.1 *Let $\Delta \in \mathbb{Z}_{<0}$ be a quadratic discriminant, and let (a, b, c) be a quadratic form of discriminant Δ. Let $n \in \mathbb{N}$, and by N_n denote the number of pairs $(x, y) \in \mathbb{Z}^2$ with $ax^2 + bxy + cy^2 \leq n$. Then we have*

$$N_n = \frac{2\pi n}{\sqrt{|\Delta|}} + O(\sqrt{n}) \,. \tag{4.2}$$

The bijection ρ from Proposition 2.1.21 shows that $N_n = \#\{(t,y) \in \mathbb{Z}^2 : t^2 - \Delta y^2 \leq 4n\}$; geometrically ρ is the transformation of the ellipse $ax^2+bxy+cy^2 = n$ to their principal axis t and y. The normal form of the ellipse is

$$\left(\frac{t}{2\sqrt{n}}\right)^2 + \left(\frac{y}{2\sqrt{n/|\Delta|}}\right)^2 \;=\; 1\,. \tag{4.3}$$

In our application we have $n \geq i_0 \geq 2^{159}$ and almost always $|\Delta| < 6000000$, hence $\frac{\sqrt{n\cdot|\Delta|}}{n} < 4.05 \cdot 10^{-21}$. Thus we approximate (4.2) by $N_n = \frac{2\pi n}{\sqrt{|\Delta|}}$. As in Section 4.1 we restrict to intervals of the form $[i_0, 2i_0 - 1]$.

Theorem 4.2.2 *Let i_0 be a positive integer and Δ a negative quadratic discriminant of class number h. If $i_1 = 2i_0 - 1$, the bit-complexity of* `generatePrimeRandomTrace`$(\Delta, [i_0, i_1])$ *is* $O(h \cdot \log^4 i_1/\sqrt{|\Delta|})$.

Proof: According to our approximation of N_n, the total number of pairs $(t,y) \in \mathbb{N}^2$ with $\frac{t^2-\Delta y^2}{4} \in [i_0, i_1]$ is equal to $\frac{1}{4} \cdot \left(\frac{2\pi\cdot i_1}{\sqrt{|\Delta|}} - \frac{2\pi\cdot(i_1+1)}{2\sqrt{|\Delta|}}\right) \approx \frac{\pi\cdot i_1}{4\sqrt{|\Delta|}}$. As estimated in the proof of Theorem 4.1.3 the number of primes in $[i_0, i_1]$ suitable for the CM-method of discriminant Δ is asymptotically equal to $\frac{i_1}{4h\log i_1}$. In Theorem 2.1.22 we showed that for a prime $p = \frac{t^2-\Delta y^2}{4}$ the pair $(t,y) \in \mathbb{N}^2$ is unique. Hence the probability of success for a randomly chosen $(t,y) \in \mathbb{N}^2$ is $\frac{\sqrt{|\Delta|}}{\pi h\cdot\log i_1}$. Thus we expect to choose $N_T := \frac{\pi h\cdot\log i_1}{\sqrt{|\Delta|}}$ pairs (t,y) before finding a prime. For each pair (t,y) we have to perform the Miller-Rabin test of bit-complexity $O(\log^3 i_1)$. Hence the total bit-complexity is $O(h \cdot \log^4 i_1/\sqrt{|\Delta|})$. □

The complexity of `generatePrimeRandomTrace` is the complexity of the algorithms of Section 4.1 divided by $\sqrt{|\Delta|}$. Hence for fixed class number h and interval $[i_0, i_1]$ the running time is reciprocally proportional to $\sqrt{|\Delta|}$; this fact is already observed by [SScK01]. However, `generatePrimeRandomTrace` turns out to be much faster in practice than `generateIsPrime` and `generateNextPrime`. The main reason is that we do not have to perform the algorithm of Cornacchia of complexity $O(\log^4 i_1)$.

Table 4.3 lists practical timings of `generatePrimeRandomTrace`. Furthermore, let N_P denote the number of chosen pairs (t,y) before the algorithm terminates, and N_T the theoretical value of the proof of Theorem 4.2.2. In Table 4.3 we compare both quantities. As in Section 4.1 we use the fundamental discriminant $\Delta = -21311$ of class number 200. The relation $\Delta \equiv 1 \bmod 8$ implies $t \equiv y \bmod 2$ for a suitable pair (t,y). However, $t \equiv y \equiv 1 \bmod 2$ yields $t^2 - \Delta y^2 \equiv 0 \bmod 8$, hence $\frac{t^2-\Delta y^2}{4}$ is even. Thus we only choose pairs (t,y) with t and y both even. Hence the theoretical value N_T in Table 4.3 is only half the value of N_T in Theorem 4.2.2. The correspondence between N_T and N_P is obvious. We remark that the running time of `generatePrimeRandomTrace` is less than 1% of `generateNextPrime`.

We show how to speed up `generatePrimeRandomTrace`. We collect necessary conditions on t and y for $\frac{t^2-\Delta y^2}{4}$ to be prime. Let f be a quadratic form of discriminant Δ, and let (x,w) be a representation of n by f, that is $n = f(x,w)$. We clearly have $\gcd(x,w)^2 \mid n$. Thus if $\gcd(x,w) > 1$, n cannot be prime. Hence we introduce the term of a proper representation.

b	t in sec.	N_P	N_T	N_P/N_T	b	t in sec.	N_P	N_T	N_P/N_T
162	0.698132	242.62	243.143	0.99785	181	0.937181	271.342	271.484	0.999475
163	0.715722	245.002	244.634	1.0015	182	0.947603	272.825	272.976	0.999448
164	0.720645	243.015	246.126	0.987362	183	0.963641	276.496	274.468	1.00739
165	0.729743	244.991	247.618	0.989394	184	0.967674	275.647	275.959	0.998867
166	0.739651	246.785	249.109	0.99067	185	0.982415	277.475	277.451	1.00009
167	0.755673	250.94	250.601	1.00135	186	0.978763	275.744	278.943	0.988532
168	0.762538	251.5	252.093	0.997648	187	1.0009	281.131	280.434	1.00248
169	0.775076	254.421	253.584	1.0033	188	1.01307	282.449	281.926	1.00185
170	0.78834	257.631	255.076	1.01002	189	1.01088	281.123	283.418	0.991903
171	0.790928	256.939	256.568	1.00145	190	1.03372	285.273	284.909	1.00128
172	0.796734	257.608	258.059	0.998251	191	1.03764	284.976	286.401	0.995025
173	0.816813	259.932	259.551	1.00147	192	1.09867	289.452	287.893	1.00541
174	0.824681	264.007	261.043	1.01135	193	1.13454	287.405	289.384	0.99316
175	0.837814	262.421	262.534	0.999569	194	1.19993	292.439	290.876	1.00537
176	0.859094	264.093	264.026	1.00025	195	1.20121	291.386	292.368	0.996643
177	0.885041	262.864	265.518	0.990006	196	1.20976	289.641	293.859	0.985644
178	0.889852	262.778	267.009	0.984154	197	1.22771	292.999	295.351	0.992035
179	0.917819	269.734	268.501	1.00459	198	1.23822	293.47	296.843	0.988638
180	0.929265	269.844	269.993	0.999449	199	1.25851	296.476	298.334	0.993771

Table 4.3: Data delivered by `generatePrimeRandomTrace`$(-21311, [2^b, 2^{b+1}-1])$. $\Delta = -21311$ is fundamental of class number 200. N_P denotes the number of trials in practice to find a prime, $N_T = \frac{\pi h \cdot \log i_1}{2\sqrt{|\Delta|}}$ the theoretical value. For each bitlength b we performed 20000 tests. The running times were measured on the SUN UltraSPARC I.

Definition 4.2.3 *Let f be a quadratic form of discriminant Δ and let n be an integer. The pair $(x,w) \in \mathbb{Z}^2$ is a* proper representation *of n by f if $f(x,w) = n$ and $\gcd(x,w) = 1$. By $\mathcal{C}(R_{\Delta,f}(n))$ we mean the set of all proper representations of n by f. Furthermore, if ρ denotes the bijection from Proposition* 2.1.21, *we set $\mathcal{C}(N_\Delta(n)) = \rho^{-1}(\mathcal{C}(R_{\Delta,f}(n)))$.*

Now let f be the main form of discriminant Δ, and let (x,w) be a proper representation of n by f. Set $(t,y) = \rho^{-1}(x,w)$ with the bijection ρ from Proposition 2.1.21. We present conditions on the integers t and y such that (x,w) properly represents n.

Theorem 4.2.4 *For $n \in \mathbb{N}$ we have*

$$\mathcal{C}(N_\Delta(n)) = \begin{cases} \{(t,y) \in N_\Delta(n) : \gcd(t,y) = 1\} \quad \cup \\ \{(t,y) \in N_\Delta(n) : 2 \,||\, t,\ 2 \mid y,\ \gcd(\frac{t}{2}, y) = 1\} \ : \quad \Delta \equiv 0 \bmod 4\ ; \\ \{(t,y) \in N_\Delta(n) : \gcd(t,y) = 1\} \quad \cup \\ \{(t,y) \in N_\Delta(n) : 2 = \gcd(t,y),\ t \not\equiv y \bmod 4\} \ : \quad \Delta \equiv 1 \bmod 4\ . \end{cases}$$

Proof: By $\mathcal{C}'(N_\Delta(n))$ we denote the right side of the theorem. Let M_1 and M_2 be subsets of $N_\Delta(n)$ and $R_{\Delta,f}(n)$, respectively, with $\rho(M_1) \subseteq M_2$ and $\rho^{-1}(M_2) \subseteq M_1$. Application of ρ^{-1} to the first inclusion yields $M_1 = \rho^{-1}(\rho(M_1)) \subseteq \rho^{-1}(M_2) \subseteq M_1$, i.e. $\rho^{-1}(M_2) = M_1$. Hence application of ρ to this equation gives $\rho(M_1) = M_2$. It therefore suffices to show $\rho(\mathcal{C}'(N_\Delta(n))) \subseteq \mathcal{C}(R_{\Delta,f}(n))$ and $\rho^{-1}(\mathcal{C}(R_{\Delta,f}(n))) \subseteq \mathcal{C}'(N_\Delta(n))$ in order to prove the theorem.

I. Let $\Delta \equiv 0 \bmod 4$. We first show $\rho(\mathcal{C}'(N_\Delta(n))) \subseteq \mathcal{C}(R_{\Delta,f}(n))$. For $(t,y) \in \mathcal{C}'(N_\Delta(n))$ let l be a positive divisor of both x and w with $(x,w) = \rho(t,y) = (\frac{t}{2}, y)$. First, let $\gcd(t,y) = 1$.

Because of $\gcd(\frac{t}{2}, y) \mid \gcd(t, y) = 1$ we get $l = 1$, i.e. $\rho(t, y) \in \mathcal{C}(R_{\Delta,f}(n))$. Now, let $2 \mid\mid t$, $2 \mid y$, and $\gcd(\frac{t}{2}, y) = 1$. The last condition implies $l = 1$, and again we have $\rho(t, y) \in \mathcal{C}(R_{\Delta,f}(n))$.
Conversely, let (x, w) be a proper representation of n by f, and set $(t, y) = \rho^{-1}(x, w) = (2x, w)$. In case of $\gcd(2x, w) = 1$ we have $\rho^{-1}(x, w) \in \mathcal{C}'(N_\Delta(n))$. Assume therefore $\gcd(2x, w) > 1$. As x and w are relatively prime the only possibility is $\gcd(2x, w) = 2$, and we get $2 \mid w$, $2 \nmid x$. This is equivalent to $2 \mid\mid t$, $2 \mid y$, and $\gcd(\frac{t}{2}, y) = 1$, and $\rho^{-1}(x, w) \in \mathcal{C}'(N_\Delta(n))$ holds.
II. Now let $\Delta \equiv 1 \bmod 4$. As Δ is odd $t^2 - \Delta y^2$ is divisible by 4 if and only if t and y have the same parity.
We first show $\rho(\mathcal{C}'(N_\Delta(n))) \subseteq \mathcal{C}(R_{\Delta,f}(n))$. Hence let $(t, y) \in \mathcal{C}'(N_\Delta(n))$. In addition, let l be a positive divisor of both x and w with $(x, w) = \rho(t, y) = (\frac{t-y}{2}, y)$. The definition of ρ shows $l \mid \frac{t-y}{2}$ and $l \mid y$, i.e. there are integers t_1, y_1 with $\frac{t-y}{2} = lt_1$ and $y = ly_1$. This is equivalent to $y = ly_1$ and $t = 2lt_1 + ly_1$, i.e. $l \mid \gcd(t, y)$. First, let $\gcd(t, y) = 1$. Thus only $l = 1$ is possible. Therefore $\rho(t, y) \in \mathcal{C}(R_{\Delta,f}(n))$ follows. Second, assume $2 = \gcd(t, y)$ and $t \not\equiv y \bmod 4$. The relation $l \mid \gcd(t, y)$ yields $l \mid 2$. Assuming $l = 2$ shows $t = 4t_1 + y$, i.e. $t \equiv y \bmod 4$. However, this contradicts the assumption $t \not\equiv y \bmod 4$.
Conversely, let (x, w) be a proper representation of n by f, and set $(t, y) = \rho^{-1}(x, w) = (2x + w, w)$. In case of $\gcd(t, y) = 1$ we are done. Let l be a non trivial positive divisor of t and y. Thus there is a $l > 1$ and integers w_1 and x_1 with $w = lw_1$ and $2x + w = lx_1$, i.e. $l \mid 2x$. Because of $\gcd(x, w) = 1$ we get $l = 2$. Hence w is even. It remains to show $t \not\equiv y \bmod 4$. Thus assume $2x + w \equiv w \bmod 4$. This shows $2x \equiv 0 \bmod 4$, hence $2 \mid x$. This is a contradiction to $\gcd(x, w) = 1$. Thus again we have $\rho^{-1}(x, w) \in \mathcal{C}'(N_\Delta(n))$. □

We get an obvious variant of algorithm `generatePrimeRandomTrace` when considering the conditions of Theorem 4.2.4. However, in Section 4.3 we will show how to further speed it up.

Finally, we discuss whether `generatePrimeRandomTrace` returns primes which are K-uniformly distributed in I for some reasonable K, say $K \geq 2^{40}$. We deduce from Figure 4.3 that this is not the case. However, when designing our optimized algorithm `findPrimeDeltaFixed` in the following section, we ensure that condition (P3) of Chapter 3 will be respected, that is the primes returned by the optimized algorithm are K-uniformly distributed for a given K.

4.3 Finding Suitable Cardinalities

In this section we extend the problem of the previous two sections as follows: Find a rational prime p of the form $p = \frac{t^2 - \Delta y^2}{4}$ such that one of the numbers $p + 1 - t$ or $p + 1 + t$ is the cardinality of a cryptographically strong elliptic curve over $\mathbb{F}_p$. We set $N_- = p + 1 - t$ and $N_+ = p + 1 + t$ in what follows.

As a result of the Sections 4.1 and 4.2 we state that the method of Section 4.2 is faster both in theory and practice. Hence, we only consider this approach. We show how to speed up algorithm `generatePrimeRandomTrace` by using congruence relations and sieving methods. The result is our very efficient algorithm `findPrimeDeltaFixed`. However, we have to distinguish three cases, depending on the residue class of Δ modulo 8: $\Delta \equiv 1 \bmod 8$, $\Delta \equiv 5 \bmod 8$, and $\Delta \equiv 0, 4 \bmod 8$. There are different lower bounds of k_0 in either case. We will show in the following subsections that we have $k_0 \geq 4$ in case of $\Delta \equiv 1 \bmod 8$ or

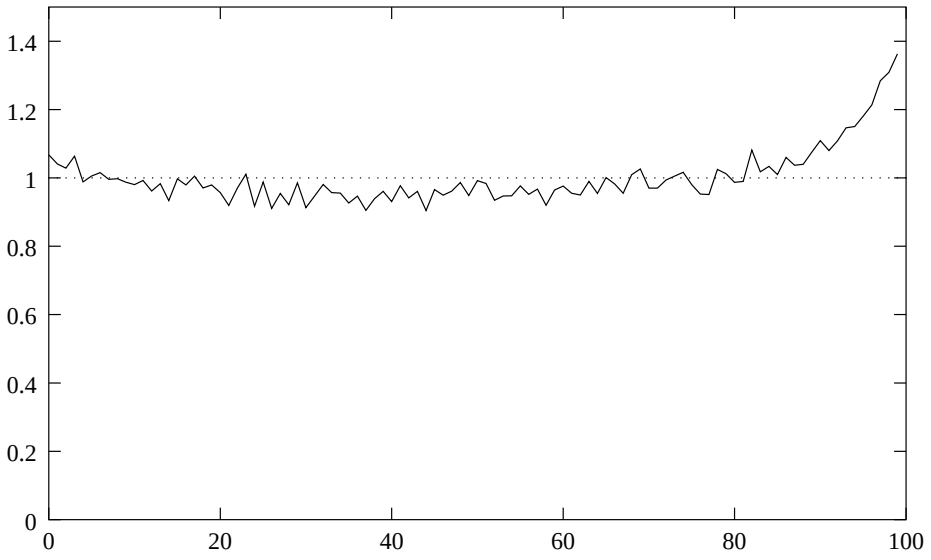


Figure 4.3: Distribution of primes returned by `generatePrimeRandomTrace`(Δ, I) for $\Delta = -21311$, $I = [2^{162}, 2^{163} - 1]$. We evaluate 20000 primes. We subdivide I in 100 subintervals charted at the abscissae. In y-direction we illustrate the ratio of the number of primes in a subinterval to the mean value 200.

$\Delta \equiv 0, 4 \bmod 16$, and $k_0 \geq 2$ in case of $\Delta \equiv 8, 12 \bmod 16$. Hence, a necessary condition for the elliptic curve to be of prime order is $\Delta \equiv 5 \bmod 8$. We remark that discriminants $\Delta \equiv 0, 4 \bmod 16$ are not fundamental. Hence, we do not take this family of discriminants into account. Our optimized subalgorithms of `findPrimeDeltaFixed` assume k_0 to be chosen minimal.

However, in order to get a theoretical bound of the complexity of `findPrimeDeltaFixed`, we first extend `generatePrimeRandomTrace` to a non-optimized algorithm, which we call `generateTwinPrimeRandomTrace`$(\Delta, [i_0, i_1])$. The naming of this algorithm will become clear soon. In addition, algorithm `generateTwinPrimeRandomTrace` restricts to curves of prime order and field discriminants, that is we assume $\Delta \equiv 5 \bmod 8$ to be fundamental while deriving the complexity of this general algorithm. It outputs a pair $(p, t) \in \mathbb{N}^2$ such that both $p = \frac{t^2 - \Delta y^2}{4}$ and $p + 1 + t$ are prime and $p \in [i_0, i_1]$. We restrict to curves of order $p + 1 + t$ as we can determine the complexity of `generateTwinPrimeRandomTrace` in that case. Nevertheless, a practical implementation will also consider curves of order $p + 1 - t$, and the running time of this implementation is bounded by the running time of our algorithm.

We explain the naming of `generateTwinPrimeRandomTrace`. Let $(t, y) \in \mathbb{N}^2$. Assume furthermore that both $p = \frac{t^2 - \Delta y^2}{4}$ and $p + 1 + t$ are prime. Thus there exists a $\pi \in \mathcal{O}_\Delta$ such that both $p = \pi\overline{\pi}$ and $p + 1 + t = (\pi + 1)(\overline{\pi + 1})$ are prime. Hence we search for a generalized twin prime $(\pi, \pi + 1) \in \mathcal{O}_\Delta$.

We next determine the complexity of `generateTwinPrimeRandomTrace`. We make use of a conjecture of [GS00]. Before stating their conjecture we have to introduce some notation. Let Δ be fundamental, $\alpha \in \mathcal{O}_\Delta$ and K be the quadratic field of discriminant Δ. A *reasonable large subset* of $\mathcal{O}_\Delta$ is a subset $\mathcal{R} = \{a + b\sqrt{\Delta} : (a, b) \in \mathbb{Z}_0^2, 2 \leq N(a + b\sqrt{\Delta}) \leq x\}$ for sufficient

Algorithm 4.4: `generateTwinPrimeRandomTrace`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 5 \bmod 8$.
An interval $[i_0, i_1] \subset \mathbb{N}$.

Output: A prime $p \in [i_0, i_1]$ and a $t \in \mathbb{N}$ such that $4p = t^2 - \Delta y^2$ with an integer y, and such that $p + 1 + t$ is prime, too.

$m \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $t \in_R \{1, ..., 2 \cdot \lfloor\sqrt{i_1}\rfloor\}$; // choose a random t
  if $\frac{4i_0 - t^2}{|\Delta|} \leq 0$ **then**
    $y \in_R \{1, ..., \lfloor\sqrt{\frac{4i_1 - t^2}{|\Delta|}}\rfloor\}$; // compute bounds of y and choose random y
  else
    $y \in_R \{\lceil\sqrt{\frac{4i_0 - t^2}{|\Delta|}}\rceil, ..., \lfloor\sqrt{\frac{4i_1 - t^2}{|\Delta|}}\rfloor\}$;
  if $t \not\equiv y \bmod 2$ **then**
    $t \leftarrow t + 1$; // ensure that $\frac{t^2 - \Delta y^2}{4} \in \mathbb{Z}$
  $n \leftarrow \frac{t^2 - \Delta y^2}{4}$;
  if `isPrime`(n, m) = `true` AND `isPrime`$(n + t + 1, m)$ = `true` **then**
    return (n, t);

large x. By $P_\alpha(K, \mathcal{R})$ we mean the number of primes $a + b\sqrt{\Delta} \in \mathcal{R}$ such that $a, b \geq 0$ and $a + b\sqrt{\Delta} + \alpha \in \mathcal{R}$. Unfortunately the notation in [GS00] is not formally defined and somehow confusing. Hence the above notation is our interpretation. Nevertheless, we will present practical results, and the conformance of these results with the theoretical value justifies our interpretation.

Conjecture 4.3.1 (Gross,Smith) *Let K be a number field. Let $\mathfrak{p}_1, \ldots, \mathfrak{p}_r$ be the complete list of prime ideals of K with norm 2, and let α be an element of $\mathcal{O}_K$ divisible by each $\mathfrak{p}_i$, and only by those ideals. (If there are no prime ideals of K with norm 2, let $\alpha = 1$.) Let $\mathcal{R}$ be a reasonable large subset of $\mathcal{O}_K$. Then*

$$P_\alpha(K, \mathcal{R}) \approx 2^r \cdot \prod_{\substack{\mathfrak{p} \\ \mathcal{N}(\mathfrak{p}) \neq 2}} \left(1 - \frac{1}{(\mathcal{N}(\mathfrak{p}) - 1)^2}\right) \cdot L(1, \chi_\Delta)^{-2} \cdot \sum_{\substack{\beta \in \mathcal{R} \\ |N(\beta) \neq 0, 1|}} \frac{1}{(\log |N(\beta)|)^2} \tag{4.4}$$

where the interpretation of the symbol $\approx$ is that for a "reasonable" sequence of increasing regions $\mathcal{R}_i$, the ratio of the two sides tends to 1.

We evaluate Equation (4.4) for discriminants $\Delta \equiv 5 \bmod 8$. Then $\chi_\Delta(2) = -1$ according to Definition 2.1.16, and 2 is inert. Thus there is no $\mathcal{O}_K$-ideal with norm 2, and $r = 0$ and $\alpha = 1$ follow. We next compute the three factors of Equation (4.4).

First, let $P_N(\Delta)$ denote the first factor. As 2 is inert we have

$$\begin{aligned} P_N(\Delta) &= \prod_{\mathfrak{p}} \left(1 - \frac{1}{(\mathcal{N}(\mathfrak{p}) - 1)^2}\right) \\ &= \prod_{\substack{p \in \mathbb{P} \\ \left(\frac{\Delta}{p}\right) = 1}} \left(1 - \frac{1}{(p-1)^2}\right)^2 \cdot \prod_{\substack{p \in \mathbb{P} \\ \left(\frac{\Delta}{p}\right) = -1}} \left(1 - \frac{1}{(p^2-1)^2}\right) \cdot \prod_{\substack{p \in \mathbb{P} \\ \left(\frac{\Delta}{p}\right) = 0}} \left(1 - \frac{1}{(p-1)^2}\right) . \end{aligned}$$

Second, we evaluate the factor $L(1,\chi_\Delta)^{-2}$ using the analytic class number formula (Equation (2.6)). It is well known that we have $u = 6$ for $\Delta = -3$, $u = 4$ for $\Delta = -4$, and $u = 2$ otherwise. As we assume $\Delta < -4$ we have $L(1,\chi_\Delta) = \frac{\pi h}{\sqrt{|\Delta|}}$.

Finally we turn to the sum. However, as stated above it is not clear over which region $\mathcal{R}$ we have to sum up, and the explanations in [GS00] are somehow confusing or false. Our interpretation is as follows. We are interested in principal prime ideals $(\pi) \subset \mathcal{O}_\Delta$ with norm in $[2,x]$ such that $(\pi+1)$ is a principal prime ideal with norm in $[2,x]$, too. Hence if π is such an element, then $\overline{\pi}$ is such an element, too. Thus when only considering generators of principal prime ideals with positive real and imaginary part, we only get exactly half of the ideals. Thus we consider this by introducing a factor 2 and setting $\mathcal{R}' = \{a+b\sqrt{\Delta} : (a,b) \in \mathbb{N}_0^2, 2 \le N(a+b\sqrt{\Delta}) \le x\}$. As in [GS00] we approximate the sum by an integral over the set $\mathcal{S} = \{a+b\sqrt{\Delta} : a,b \ge 0, 2 \le N(a+b\sqrt{\Delta}) \le x\}$.

$$\begin{aligned}
\sum_{\substack{\beta\in\mathcal{R}\\ |N(\beta)\neq 0,1|}} \frac{1}{(\log|N(\beta)|)^2} &= 2\cdot\sum_{\beta\in\mathcal{R}'}\frac{1}{(\log|N(\beta)|)^2}\\
&\approx 2\cdot\int_{\mathcal{S}}\frac{da\,db}{(\log(a^2-\Delta b^2))^2}\\
&= 2\cdot\int_{\sqrt{2}}^{\sqrt{x}}\int_0^{\frac{\pi}{2}}\frac{\frac{r}{\sqrt{|\Delta|}}dr\,d\varphi}{(\log(r^2))^2} \qquad (4.5)\\
&= 2\cdot\frac{\pi}{2\sqrt{|\Delta|}}\int_{\sqrt{2}}^{\sqrt{x}}\frac{2rdr}{2(\log(r^2))^2} \qquad (4.6)\\
&= \frac{\pi}{2\sqrt{|\Delta|}}\int_2^x\frac{dy}{(\log y)^2}\,. \qquad (4.7)
\end{aligned}$$

In Equation (4.5) we make use of the bijective transformation $\phi : [\sqrt{2},\sqrt{x}]\times[0,\frac{\pi}{2}] \to \mathcal{S}$, $(r,\varphi)\mapsto(r\cos\varphi, \frac{r}{\sqrt{|\Delta|}}\sin\varphi)$ of functional determinant $\frac{r}{\sqrt{|\Delta|}}$. Next in (4.6) we substitute r^2 by y.

Now let $i_1 = 2i_0 - 1$. We develop a closed formula for the probability that the algorithm succeeds for a randomly chosen pair (t,y). We denote this probability by W. From Section 4.2 we already know that there are $1/4\cdot 2\pi i_0/\sqrt{|\Delta|}$ pairs $(t,y)\in\mathbb{N}_0^2$ such that $(t^2-\Delta y^2)/4 \in I$. Hence using the above estimation on $P_1(K,\mathcal{R})$ and remembering that for a prime $p = (t^2-\Delta y^2)/4$ the pair (t,y) is unique we deduce

$$\begin{aligned}
W &= \frac{P_N(\Delta)\cdot\left(\frac{\pi h}{\sqrt{|\Delta|}}\right)^{-2}\cdot\frac{\pi}{2\sqrt{|\Delta|}}\int_{i_0}^{i_1}\frac{dy}{(\log y)^2}}{\frac{1}{4}\cdot\frac{2\pi i_0}{\sqrt{|\Delta|}}} \qquad (4.8)\\
&= \frac{P_N(\Delta)\cdot|\Delta|\cdot\int_{i_0}^{i_1}\frac{dy}{(\log y)^2}}{\pi^2 h^2 i_0}\,. \qquad (4.9)
\end{aligned}$$

In [Bro93], p.774, formula 470, we find $\int_{i_0}^{i_1}\frac{dy}{(\log y)^2} = -\frac{x}{\log x}\Big|_{i_0}^{i_1} + \int_{i_0}^{i_1}\frac{dy}{\log y} = \frac{i_0}{\log i_0} - \frac{i_1}{\log i_1} + \mathrm{li}(i_1) - \mathrm{li}(i_0)$. We will use this equation to compute the integral in practice. However, to

estimate the complexity we make use of a formula used by [GS00] and [SScK01]: Both approximate the antiderivative by $\frac{x}{\log^2 x}\sum_{k=1}^{N}\frac{k!}{\log^{k-1} x}$ for a certain $N \in \mathbb{N}$. Thus, asymptotically we have $\int_{i_0}^{i_1}\frac{dy}{(\log y)^2} \sim \frac{i_1}{(\log i_1)^2} - \frac{i_0}{(\log i_0)^2} \sim \frac{i_1}{2(\log i_1)^2}$. Before determining the complexity of `generateTwinPrimeRandomTrace` we prove the following lemma.

Lemma 4.3.2 *Let $\Delta \equiv 5 \bmod 8$ be an imaginary quadratic field discriminant. Then*

$$\frac{8}{9}\prod_{\substack{p\in\mathbb{P}\\ p\geq 3}}\left(1-\frac{1}{(p-1)^2}\right)^2 \;\leq\; P_N(\Delta) \;\leq\; \frac{8}{9}\prod_{\substack{p\in\mathbb{P}\\ p\geq 3}}\left(1-\frac{1}{(p^2-1)^2}\right). \tag{4.10}$$

Proof: As $p=2$ is inert in $\mathbb{Q}(\sqrt{\Delta})$ we have a factor $1-\frac{1}{(2^2-1)^2}=\frac{8}{9}$ in $P_N(\Delta)$. Let $p \geq 3$. We prove $\left(1-\frac{1}{(p-1)^2}\right)^2 \leq 1-\frac{1}{(p-1)^2} \leq 1-\frac{1}{(p^2-1)^2}$. As $0 < 1-\frac{1}{(p-1)^2} < 1$, the first inequality is obvious. The second one is equivalent to $(p-1)^2 \leq (p^2-1)^2$, which again is obvious. Using the above formula for $P_N(\Delta)$ completes the proof. □

We remark that the lower bound corresponds to the case where every prime $p \geq 3$ splits completely in $\mathbb{Q}(\sqrt{\Delta})$, while the upper bound corresponds to the case where every prime $p \geq 3$ is inert in $\mathbb{Q}(\sqrt{\Delta})$. We computed the bounds of Lemma 4.3.2 up to $p = 15000179$ using a floating point precision of 100. Using this approximation, the first 10 significant digits of the bounds are $0.3873898902 \leq P_N(\Delta) \leq 0.8729859532$. Next we state the complexity of `generateTwinPrimeRandomTrace`.

Theorem 4.3.3 *Let i_0 be a positive integer, and let $\Delta \equiv 5 \bmod 8$ be an imaginary quadratic field discriminant. Furthermore, let h denote the class number of $\mathcal{O}_\Delta$. If we set $i_1 = 2i_0 - 1$, the bit-complexity of* `generateTwinPrimeRandomTrace`$(\Delta, [i_0, i_1])$ *is $O(h^2 \cdot \log^5 i_1/|\Delta|)$.*

Proof: As above denote by W the probability that `generateTwinPrimeRandomTrace` succeeds for a randomly chosen pair (t, y). Equation (4.9) and the approximation of the integral yield $W = \frac{P_N(\Delta)\cdot|\Delta|}{\pi^2 h^2\cdot(\log i_1)^2}$. Hence we expect to choose $\frac{\pi^2 h^2\cdot(\log i_1)^2}{P_N(\Delta)\cdot|\Delta|}$ pairs (t, y) before `generateTwinPrimeRandomTrace` terminates. The factor $P_N(\Delta)$ varies with the discriminant. However, we have $P_N(\Delta) \in [0.38, 0.88]$, as we have shown in Lemma 4.3.2. For each pair (t, y) we have to perform at most 2 Miller-Rabin tests, each of bit-complexity $O(\log^3 i_1)$ ([Coh95], Section 8.2, p. 415). Hence the total bit-complexity is $O(h^2 \cdot \log^5 i_1/|\Delta|)$. □

Finally, we show that Formula (4.9) well fits with practical tests. For example, we set $\Delta = -356131$. We then have $h(\Delta) = 200$. For $i_0 = 2^b$ with $159 \leq b \leq 180$ we counted the number of randomly chosen pairs (t, y) in our implementation of `generateTwinPrimeRandomTrace`. For each b we performed 500 tests. The comparison of our practical data to the theoretical value of Equation (4.9) is shown in Figure 4.4. The coincidence is obvious. We have $P_N(-356151) = 0.68386$.

We next explain our efficient algorithm `findPrimeDeltaFixed`. Depending on the input k_0, `findPrimeDeltaFixed` first chooses an appropriate fundamental discriminant Δ of class number at least h_0. More precisely, as explained above, if $k_0 = 1$ and $k_0 \in \{1; 2; 3\}$ we have to

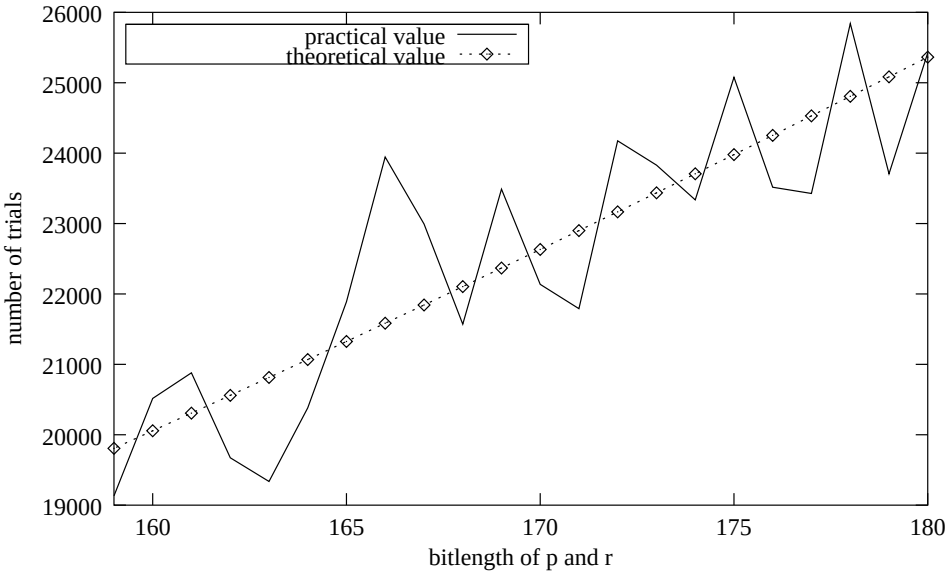


Figure 4.4: Comparison of the number of randomly chosen pairs (t, y) in algorithm `generateTwinPrimeRandomTrace` in both theory and practice. For each bitlength we performed 500 tests. We set $\Delta = -356131$ with $h(\Delta) = 200$.

take care of $\Delta \equiv 5 \bmod 8$ and $\Delta \not\equiv 1 \bmod 8$, respectively. However, if $\Delta \equiv 1 \bmod 8$ and $3 \nmid \Delta$ we may use the class polynomial W introduced in Section 2.2.3 to generate the ring class field. This polynomial has rather small coefficients and hence may be computed efficiently. Thus if $k_0 = 4$, we only take discriminants $\Delta \equiv 1 \bmod 8$ into account. The user may specify such a discriminant with $h(\Delta) \geq h_0$; if no discriminant is specified, using our database `delta_h`, the algorithm chooses the largest discriminant $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$ of class number h_0. In the formal description of the algorithm we denote this proceeding as "Get $\Delta \equiv 1 \bmod 8$;". In addition, in order to distinguish the different input k_0, if $k_0 \in \{2; 3\}$ we only consider discriminants $\Delta \equiv 8, 12 \bmod 16$. The discriminant is again chosen to be maximal of class number h_0. Again we ensure $3 \nmid \Delta$ to make use of the polynomial G for generating the Hilbert class field. Finally, if $k_0 = 1$, we have to ensure $\Delta \equiv 5 \bmod 8$.

Once Δ is chosen, `findPrimeDeltaFixed` determines an interval $[i_0, i_1]$ with $i_0 = 2^b$, $i_1 = 2^{b+1} - 1$, and $b = \lfloor \log_2 r_0 k_0 \rfloor$. It then invokes the appropriate subalgorithm. All subalgorithms determine a prime $p \in [i_0, i_1]$ and the cardinality of a cryptographically strong elliptic curve over $\mathbb{F}_p$ having $\mathcal{O}_\Delta$ as endomorphism ring. Hence, r will be of bitlength at least $\lfloor \log_2 r_0 \rfloor$. The algorithm is very efficient in practice. For instance, if $b = 162$ and $\Delta \equiv 1 \bmod 8$ of class number 200, the algorithm terminates after 0.2 seconds on the Pentium III. We will see in what follows that the primes returned by `findPrimeDeltaFixed` are K-uniformly distributed in $[i_0, i_1]$ with $K = 2^{40}$.

4.3.1 Discriminants $\Delta \equiv 1 \bmod 8$

We discuss in detail our very efficient algorithm `findPrime1Mod8`$(r_0, k_0, \Delta, [i_0, i_1])$. We present running times proving the efficiency in practice. For instance, when searching for a crypto-

Algorithm 4.5: `findPrimeDeltaFixed`(r_0, k_0, h_0)

Input: Positive integers r_0, k_0, and h_0.

Output: A fundamental imaginary quadratic discriminant Δ with $h(\Delta) \geq h_0$.
Rational primes p and r, and a positive integer k with $r \geq r_0$, $k \leq k_0$, and $\lfloor \log_2 p \rfloor = \lfloor \log_2 rk \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$, such that a cryptographically strong elliptic curve E over $\mathbb{F}_p$ exists with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.

if $k_0 \geq 4$ **then**
  Get $\Delta \equiv 1 \bmod 8$; //Explanation: see Page 60
  $b \leftarrow \lfloor \log_2 4r_0 \rfloor$; $i_0 \leftarrow 2^b$; $i_1 \leftarrow 2^{b+1} - 1$;
  `findPrime1Mod8`$(r_0, 4, \Delta, [i_0, i_1])$;
else if $k_0 \geq 2$ **then**
  Get $\Delta \equiv 8, 12 \bmod 16$; //Explanation: see Page 60
  $b \leftarrow \lfloor \log_2 r_0 k_0 \rfloor$; $i_0 \leftarrow 2^b$; $i_1 \leftarrow 2^{b+1} - 1$;
  `findPrime0Mod4`$(r_0, k_0, \Delta, [i_0, i_1])$;
else
  Get $\Delta \equiv 5 \bmod 8$; //Explanation: see Page 60
  $b \leftarrow \lfloor \log_2 r_0 \rfloor$; $i_0 \leftarrow 2^b$; $i_1 \leftarrow 2^{b+1} - 1$;
  `findPrime5Mod8`$(r_0, 1, \Delta, [i_0, i_1])$;

graphically strong elliptic curve over a 162-bit field, the running time of `findPrime1Mod8` is about 0.2 seconds on the Pentium III. Finally, we give evidence that for fixed discriminant its bit-complexity is proportional to $\log^4 i_1$. Hence, the complexity of Theorem 4.3.3 seems to be an upper bound of the complexity of `findPrime1Mod8`. We are not aware of any comparable fast algorithm. We first refine the conditions on t and y to properly represent an integer. The following proposition is based on Theorem 4.2.4.

Proposition 4.3.4 *Let $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 1 \bmod 8$. For an odd integer n we have*

$$\mathcal{C}(N_\Delta(n)) = \{(t, y) \in N_\Delta(n) : 2 = \gcd(t, y),\ t \not\equiv y \bmod 4\}\ .$$

Proof: Let $(t, y) \in \mathcal{C}(N_\Delta(n))$. According to Theorem 4.2.4 it suffices to show $\gcd(t, y) > 1$. Thus assume $\gcd(t, y) = 1$. As usual we have to ensure $t \equiv \Delta y \bmod 2$. The assertion $\Delta \equiv 1 \bmod 8$ shows $t \equiv y \bmod 2$. Furthermore, as we assume $\gcd(t, y) = 1$, we have that both t and y are odd. Thus $t^2 \equiv y^2 \equiv 1 \bmod 8$ and $4n = t^2 - \Delta y^2 \equiv 1 - 1 \cdot 1 \equiv 0 \bmod 8$. This shows that n is even, which contradicts the assumption that n is odd. □

As both t and y are even we set $t_{1/2} = t/2$ and $y_{1/2} = y/2$ in what follows. Thus for odd n we have $\mathcal{C}(N_\Delta(n)) = \{(2t_{1/2}, 2y_{1/2}) \in N_\Delta(n) : 1 = \gcd(t_{1/2}, y_{1/2}),\ t_{1/2} \not\equiv y_{1/2} \bmod 2\}$. In [BB00] we prove the following proposition.

Proposition 4.3.5 *If $\Delta \equiv 1 \bmod 8$ and $t_{1/2}, y_{1/2}$ are positive integers such that $p = t_{1/2}^2 - \Delta y_{1/2}^2$ is a prime, then $N_- \equiv N_+ \equiv 0 \bmod 4$. Furthermore, if $\frac{N_+}{4}$ is prime, then we have $(t_{1/2} \bmod 4, y_{1/2} \bmod 4) \in \{(1, 0), (3, 2)\}$, and if $\frac{N_-}{4}$ is prime, $(t_{1/2} \bmod 4, y_{1/2} \bmod 4) \in \{(1, 2), (3, 0)\}$ follows.*

Hence we have $k \geq 4$, and we set $k = k_0 = 4$ in what follows. The basic idea of our variant of `generatePrimeRandomTrace` is as follows: In `generatePrimeRandomTrace` we randomly choose independent pairs $(t_{1/2}, y_{1/2})$ until a suitable one is found. However, given an initial pair $(t_{1/2}, y_{1/2})$, we can collect congruence relations on $(t_{1/2} \bmod m, y_{1/2} \bmod m)$ for small moduli m such that $t_{1/2}^2 - \Delta y_{1/2}^2$ and either $\frac{N_-}{4}$ or $\frac{N_+}{4}$ may be prime. To be more precise for fixed odd modulus m we are only interested in pairs $(t_{1/2} \bmod m, y_{1/2} \bmod m)$, which satisfy $t_{1/2}^2 - \Delta y_{1/2}^2 \not\equiv 0 \bmod m$ and either $\frac{N_-}{4} \not\equiv 0 \bmod m$ or $\frac{N_+}{4} \not\equiv 0 \bmod m$. If this is the case, we say that this pair *satisfies the congruence relation* for Δ and m. For fixed discriminant Δ we say that a pair satisfies the congruence relations, if it satisfies the congruence relation for each evaluated m. Basing on the tables in Section A.1.1 of the appendix, we evaluate the moduli $m \in \{3, 4, 5, 7\}$. Furthermore, we will show how to modify $t_{1/2}$ respectively $y_{1/2}$ such that the modified pair will satisfy the congruence relations, too. Hence we do not choose independent pairs $(t_{1/2}, y_{1/2})$. In addition, we will introduce sieving methods to minimize the amount of computation of multiprecision numbers. Finally, to avoid distinguishing several cases and to speed up sieving we fix an initial pair $(t_{1/2}, y_{1/2})$ and leave $y_{1/2}$ unchanged, that is we only modify $t_{1/2}$.

We first explore the modulus $m = 3$. In addition assume $\Delta \equiv 1 \bmod 3$. The appropriate table in Section A.1.1 shows that $\frac{N_-}{4}$ can only be prime in the case $(t_{1/2} \bmod 3, y_{1/2} \bmod 3) = (2, 0)$. Furthermore, in order for $\frac{N_+}{4}$ to be prime, we have to ensure $(t_{1/2} \bmod 3, y_{1/2} \bmod 3) = (1, 0)$. However, if $\Delta \equiv 0, 2 \bmod 3$ both choices are suitable, too. Hence, again to avoid distinguishing several cases for different values $\Delta \bmod 3$ we only consider the pairs $(t_{1/2} \bmod 3, y_{1/2} \bmod 3) = (2, 0)$ and $(t_{1/2} \bmod 3, y_{1/2} \bmod 3) = (1, 0)$ for N_- and N_+, respectively.

Next we consider the modulus $m = 5$. The tables in Section A.1.1 show that the crucial values of Δ are $\Delta \equiv 1, 4 \bmod 5$. More precisely, if $\Delta \bmod 5 \in \{1, 4\}$ there are less pairs $(t_{1/2} \bmod 5, y_{1/2} \bmod 5)$ satisfying the congruence relations for Δ and 5 than in case of $\Delta \bmod 5 \in \{0, 2, 3\}$. In all, for either $\Delta \equiv 1 \bmod 5$ or $\Delta \equiv 4 \bmod 5$, there are 18 such pairs. However, as mentioned above, we have a fixed value $y_{1/2}$. Obviously, there are 6 pairs of the form $(t_{1/2} \bmod 5, 0)$ for either discriminant, and for fixed $y_{1/2} \bmod 5$ this is maximal. Furthermore, these pairs appear in each table of Section A.1.1. Hence, we set $y_{1/2}$ such that $y_{1/2} \equiv 0 \bmod 5$. Then if $\frac{N_-}{4}$ is prime, $t_{1/2} \bmod 5 \in \{2, 3, 4\}$ follows. In addition, if $\frac{N_+}{4}$ is prime, we have $t_{1/2} \bmod 5 \in \{1, 2, 3\}$.

Analogously we proceed with the modulus 7. Values $\Delta \bmod 7$ with few pairs $(t_{1/2} \bmod 7, y_{1/2} \bmod 7)$ satisfying the congruence relations for Δ and 7 are $\Delta \bmod 7 \in \{1, 2, 4\}$. In either case there are 5 pairs of the form $(t_{1/2} \bmod 7, 0)$ yielding suitable values N_- respectively N_+, and for fixed $y_{1/2} \bmod 7$ this is maximal. Furthermore, these pairs appear in all tables of Section A.1.1. Hence we set $y_{1/2} \equiv 0 \bmod 7$. We consider N_- if and only if $t_{1/2} \bmod 7 \in \{2, 3, 4, 5, 6\}$, and we consider N_+ if and only if $t_{1/2} \bmod 7 \in \{1, 2, 3, 4, 5\}$. Table 4.4 summarizes the congruence conditions of this section.

We introduce a bit b_4 to keep track of the difference $t_{1/2} - y_{1/2} \bmod 4$: We set $b_4 = 0$ if this difference is equal to 1, and we set $b_4 = 1$ if $t_{1/2} - y_{1/2} \bmod 4 = 3$. In addition we set $t_3 = t_{1/2} \bmod 3$, $t_5 = t_{1/2} \bmod 5$, and $t_7 = t_{1/2} \bmod 7$. Next, assume $(t_{1/2}, y_{1/2})$ to be a pair respecting the requirements of Table 4.4, that is

$$(b_4, t_3, t_5, t_7) \in \begin{cases} \{0\} \times \{1\} \times \{1, 2, 3\} \times \{1, 2, 3, 4, 5\} & \text{in case of } N_+ \,, \\ \{1\} \times \{2\} \times \{2, 3, 4\} \times \{2, 3, 4, 5, 6\} & \text{in case of } N_- \,. \end{cases}$$

m	$y_{1/2} \bmod m$	N_-	N_+
4	0	$t_{1/2} \equiv 3 \bmod 4$	$t_{1/2} \equiv 1 \bmod 4$
4	2	$t_{1/2} \equiv 1 \bmod 4$	$t_{1/2} \equiv 3 \bmod 4$
3	0	$t_{1/2} \equiv 2 \bmod 3$	$t_{1/2} \equiv 1 \bmod 3$
5	0	$t_{1/2} \bmod 5 \in \{2,3,4\}$	$t_{1/2} \bmod 5 \in \{1,2,3\}$
7	0	$t_{1/2} \bmod 7 \in \{2,3,4,5,6\}$	$t_{1/2} \bmod 7 \in \{1,2,3,4,5\}$

Table 4.4: Congruence conditions on $t_{1/2}$ and $y_{1/2}$ for $\frac{N_-}{4}$ respectively $\frac{N_+}{4}$ to be prime. Δ is congruent 1 mod 8.

If we are not successful for this pair, we modify it as follows: Due to $m = 4$ we have to preserve the odd parity of $t_{1/2}$. Hence we successively increase $t_{1/2}$ by 2 until the modified pair $(t'_{1/2}, y_{1/2})$ satisfies the congruence conditions, too, that is we search the minimal $c \in \mathbb{N}$ such that both $(t_{1/2}, y_{1/2})$ and $(t_{1/2} + 2c, y_{1/2})$ respect the conditions of Table 4.4. In Table 4.5 we list the values of c for all initial vectors (b_4, t_3, t_5, t_7).

c	(b_4, t_3, t_5, t_7)
17	$(0,1,3,4)$
12	$(1,2,3,6)$, $(0,1,3,5)$
11	$(0,1,1,1)$, $(0,1,1,2)$, $(0,1,2,4)$, $(0,1,2,5)$
7	$(1,2,2,2)$, $(1,2,2,3)$, $(1,2,3,2)$, $(1,2,3,3)$, $(1,2,3,4)$, $(1,2,3,5)$, $(1,2,4,4)$, $(1,2,4,5)$
6	$(1,2,2,4)$, $(1,2,2,5)$, $(1,2,2,6)$, $(0,1,1,3)$, $(0,1,1,4)$, $(0,1,1,5)$
5	$(0,1,2,1)$, $(0,1,2,2)$, $(0,1,2,3)$, $(0,1,3,1)$, $(0,1,3,2)$, $(0,1,3,3)$
1	$(1,2,4,2)$, $(1,2,4,3)$, $(1,2,4,6)$

Table 4.5: Minimal values c such that both (b_4, t_3, t_5, t_7) and $(b_4 + c \bmod 2, t_3 + 2c \bmod 3, t_5 + 2c \bmod 5, t_7 + 2c \bmod 7)$ respect the congruence conditions of Table 4.4.

We remark that if $(t_{1/2}, y_{1/2})$ is initialized such that (b_4, t_3, t_5, t_7) appears in Table 4.5, we have $b_4 = 0 \Leftrightarrow t_3 = 1$ and $b_4 = 1 \Leftrightarrow t_3 = 2$. Hence we do not take b_4 into account in what follows. We introduce a method `cycles1Mod8`(t_3, t_5, t_7). Let (t_3, t_5, t_7) be such that the corresponding (b_4, t_3, t_5, t_7) is an element of Table 4.5. Then `cycles1Mod8`(t_3, t_5, t_7) returns the value c of Table 4.5. For instance, we have `cycles1Mod8`$(1, 3, 4) = 17$.

Next, we show how to use sieving methods. The idea is similar to that mentioned in Section 4.1.2. Let p_i denote the i-th odd prime. The primes 3, 5, and 7 are already covered by our congruence conditions. Hence, for sieving we only consider the primes $p_4 = 11$ to p_M for some $M \geq 4$. For $0 \leq i \leq M-4$ we store the residues of $t_{1/2}$ and $t_{1/2}^2 - \Delta y_{1/2}^2$ modulo p_{i+4} in arrays $t[\,]$ and $p[\,]$, respectively, that is we set $t[i] = t_{1/2} \bmod p_{i+4}$ and $p[i] = t_{1/2}^2 - \Delta y_{1/2}^2 \bmod p_{i+4}$ for $0 \leq i \leq M-4$, respectively. We tested different values of M in practice, and according to our experience the choice $M = 24$ is optimal, that is our sieving primes are the rational primes in the interval $[11, 97]$. Thus a necessary condition for $t_{1/2}^2 - \Delta y_{1/2}^2$ to be prime is $p[i] \neq 0$ for all $0 \leq i \leq M-4$. In addition, necessary conditions for $\frac{N_-}{4}$ and $\frac{N_+}{4}$ to be prime are $p[i] + 1 - 2t[i] \not\equiv 0 \bmod p_{i+4}$ and $p[i] + 1 + 2t[i] \not\equiv 0 \bmod p_{i+4}$, respectively.

Algorithm 4.6: `findPrime1Mod8`$(r_0, 4, \Delta, [i_0, i_1])$

Input: Positive integers r_0 and $k_0 = 4$.
A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 1 \bmod 8$.
An interval $[i_0, i_1] \subset \mathbb{N}$ with $i_1 - i_0 + 1 \geq 2^{159}$.
Output: The discriminant Δ.
A prime $p \in [i_0, i_1]$.
A prime $r \geq r_0$ such that there exists a cryptographically strong elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = 4r$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
The cofactor $k = 4$.

$K \leftarrow 2^{40}$; $l \leftarrow \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor$; $T \leftarrow 3000$; $M \leftarrow 24$;
$m_R \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $s \in_R \{0, \ldots K-1\}$; $v \leftarrow i_0 + sl$; $w \leftarrow v + l - 1$; //choose a random subinterval
  $\tau \in_R \{1, ..., \lfloor \sqrt{v} \rfloor\}$; $\mu \leftarrow \left\lceil \sqrt{(v - \tau^2)/|\Delta|} \right\rceil$;
  $t_{1/2} \leftarrow \min\{x \geq \tau : x \equiv 1 \bmod 210\}$;
  $y_{1/2} \leftarrow \min\{x \geq \mu : x \equiv 0 \bmod 210, t_{1/2} - x \equiv 1 \bmod 4\}$;
  $p \leftarrow t_{1/2}^2 - \Delta y_{1/2}^2$;
  $c \leftarrow 0$; $c_r \leftarrow 0$; $j \leftarrow 0$;
  $t_3 \leftarrow 1$; $t_5 \leftarrow 1$; $t_7 \leftarrow 1$;
  for $i = 0$ to $M - 4$ **do**
    $p[i] \leftarrow p \bmod p_{i+4}$; //initialize $p[\,]$, i.e. $p \bmod 11, \ldots, p \bmod 97$
    $t[i] \leftarrow t_{1/2} \bmod p_{i+4}$; // initialize $t[\,]$, i.e. $t_{1/2} \bmod 11, \ldots, t_{1/2} \bmod 97$
  while $j < T$ AND $p \in [v, w]$ **do**
    if $p[i] \neq 0$ for all $0 \leq i \leq M - 4$ **then**
      $b_m \leftarrow$ `false`; $b_p \leftarrow$ `false`; // booleans to keep track of loops
      if $t_3 = 2$ AND $p[i] + 1 - 2t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        $p \leftarrow p + 4c_r t_{1/2} + 4c_r^2$; $t_{1/2} \leftarrow t_{1/2} + 2c_r$; $c_r \leftarrow 0$; $b_m \leftarrow$ `true`;
        if $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true` **then**
          $b_p \leftarrow$ `true`;
          if `isStrong`$(r_0, 4, p, p + 1 - 2t) \neq 0$ **then**
            return($\Delta, p, (p + 1 - 2t)/4, 4$);
      if $t_3 = 1$ AND $p[i] + 1 + 2t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        if $b_m =$ `false` **then**
          $p \leftarrow p + 4c_r t_{1/2} + 4c_r^2$; $t_{1/2} \leftarrow t_{1/2} + 2c_r$; $c_r \leftarrow 0$;
        if $b_p =$ `true` OR ($b_m =$ `false` AND $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true`) **then**
          if `isStrong`$(r_0, 4, p, p + 1 + 2t) \neq 0$ **then**
            return($\Delta, p, (p + 1 + 2t)/4, 4$);
    $c \leftarrow$ `cycles1Mod8`(t_3, t_5, t_7); $c_r \leftarrow c_r + c$
    $t_3 \leftarrow t_3 + 2c \bmod 3$; $t_5 \leftarrow t_5 + 2c \bmod 5$; $t_7 \leftarrow t_7 + 2c \bmod 7$;
    for $i = 0$ to $M - 4$ **do**
      $p[i] \leftarrow p[i] + 4ct[i] + 4c^2 \bmod p_{i+4}$; $t[i] \leftarrow t[i] + 2c \bmod p_{i+4}$;
    $j \leftarrow j + 1$;

We are now able to introduce our optimized algorithm `findPrime1Mod8`$(r_0, 4, \Delta, [i_0, i_1])$. Its input are the security parameters r_0 and $k_0 = 4$. In addition, it requires a discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 1 \bmod 8$ and positive integers $i_0 < i_1$. Its output is the discriminant Δ and a rational prime $p \in [i_0, i_1]$. Furthermore, it returns a rational prime $r \geq r_0$ such that there exists a cryptographically strong elliptic curve E defined over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = 4r$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$. Finally, `findPrime1Mod8` returns the cofactor 4.

In Section 4.2 we stated that our generic algorithm `generatePrimeRandomTrace` returns primes that are not K-uniformly distributed in $[i_0, i_1]$ for some reasonable K, i.e. $K \geq 2^{40}$. In `findPrime1Mod8` we set $K = 2^{40}$. Hence, our subintervals are of length $l := \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor$. In order for this quantity to be large enough we assume $[i_0, i_1]$ to be sufficiently large, say $i_1 - i_0 + 1 \geq 2^{159}$, which is the case in cryptographic applications. We then choose a random subinterval $[v, w] = [i_0 + sl, i_0 + (s+1)l - 1]$ by choosing a random integer $s \in [0, \ldots, K-1]$. We ensure $p \in [v, w]$; hence, the primes p returned by `findPrime1Mod8` are K-uniformly distributed in $[i_0, i_0 + K \cdot \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor]$. From a practical point of view, this is equivalent to a K-uniformly distribution in $[i_0, i_1]$.

Next, we explain how we initialize $(t_{1/2}, y_{1/2})$. First, we randomly choose a lower bound of $t_{1/2}$, which we denote τ. Once, τ is chosen, we search for a lower bound μ of $y_{1/2}$ such that $\tau^2 - \Delta\mu^2 \approx v$. More precisely, let $\tau \in \{0, \ldots \sqrt{v}\}$ be a randomly chosen integer. Set $\mu = \left\lceil \sqrt{(v - \tau^2)/|\Delta|} \right\rceil$; hence we have $\tau^2 - \Delta(\mu - 1)^2 < v \leq \tau^2 - \Delta\mu^2$. The initial pair $(t_{1/2}, y_{1/2})$ has to respect the congruence conditions of Table 4.4. For efficiency reasons we initialize $t_{1/2} \equiv 1 \bmod q$ for $q \in \{2, 3, 5, 7\}$. Hence, we set $t_{1/2} = \min\{x \geq \tau : x \equiv 1 \bmod 210\}$. Furthermore, the initial value of $y_{1/2}$ has to be an even number divisible by 105. As we initialize $t_{1/2} \equiv 1 \bmod 3$ we have the additional boundary condition $t_{1/2} - y_{1/2} \equiv 1 \bmod 4$. Thus we initialize $y_{1/2} = \min\{x \geq \mu : x \equiv 0 \bmod 210, t_{1/2} - x \equiv 1 \bmod 4\}$.

Furthermore, making use of our sieving primes $p[\,]$, we are able to reduce the computational amount with multiprecision integers. We keep track of the current values of p and $t_{1/2}$ by introducing a counter c_r. More precisely, let $(t_{1/2}, y_{1/2})$ be a pair such that $t_{1/2}^2 - \Delta y_{1/2}^2 \not\equiv 0 \bmod p_{i+4}$ for all $0 \leq i \leq M - 4$. We set $p = t_{1/2}^2 - \Delta y_{1/2}^2$ and $c_r = 0$. If `findPrime1Mod8` does not terminate for the current pair $(t_{1/2}, y_{1/2})$, we increase c_r successively by $c =$ `cycles1Mod8`(t_3, t_5, t_7) until $p[i] \neq 0$ and either $p[i] + 1 - 2t[i] \bmod p_{i+4} \neq 0$ or $p[i] + 1 + 2t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$. Hence c_r is the minimal number of increasing steps of $t_{1/2}$ by 2 such that no sieving prime indicates a failure. If this turns out to be the case, we set $p' = (t_{1/2} + 2c_r)^2 - \Delta y_{1/2}^2 = p + 4c_r t_{1/2} + 4c_r^2$. In addition we adapt $t_{1/2}$ by setting $t'_{1/2} = t_{1/2} + 2c_r$.

Beyond it we introduce two booleans b_p and b_m. If no sieving prime of $p[\,]$ indicate the compositeness of the current value of p, we initialize b_p and b_m with `false`, respectively. b_m is set to `true` if the current value of p is built in order to check whether $p + 1 - 2t_{1/2}$ is the order of a cryptographically strong elliptic curve over $\mathbb{F}_p$. In addition, if p turns out to be in $[v, w]$ and if p passes the Miller-Rabin primality tests, then b_p is set to `true`. b_p is used to avoid performing the primality test twice on the same value of p. In addition, b_m stores whether p has to be built in the if-loop for $p + 1 + 2t_{1/2}$.

Finally, we initialize a new pair $(t_{1/2}, y_{1/2})$ if we are not successful for T adjacent pairs respecting the requirements of Table 4.4. We tested several choices of T, and $T = 3000$ seems to be optimal. Hence we can state `findPrime1Mod8` as Algorithm 4.6.

We demonstrate the efficiency of our algorithm in practice. For all $160 \leq b \leq 500$, b divisible by 10, we measured the timings of `findPrime1Mod8`$(2^{b-3}, 4, -21311, [2^{b-1}, 2^b - 1])$ on the Pentium III. For each bitlength b we performed 1000 tests. It turns out that `findPrime1Mod8` is very fast in practice. For example, $b = 160$ yields an average running time of 0.2 seconds. Furthermore, even if $b = 500$, `findPrime1Mod8` is expected to terminate after about 20 seconds. In Figure 4.5 the running time is plotted as a function of b. In addition, Figure 4.5 shows the function $T_0 \cdot b^4/b_0^4$, where T_0 is the running time for $b_0 = 160$. Hence, Figure 4.5 indicates that for fixed discriminant Δ the complexity of `findPrime1Mod8`$(r_0, 4, \Delta, [i_0, i_1])$ is proportional to b^4 and hence to $\log^4 i_1$. This result well fits to Theorem 4.3.3, as the latter complexity is assumed to be an upper bound of the complexity of our optimized subalgorithms.

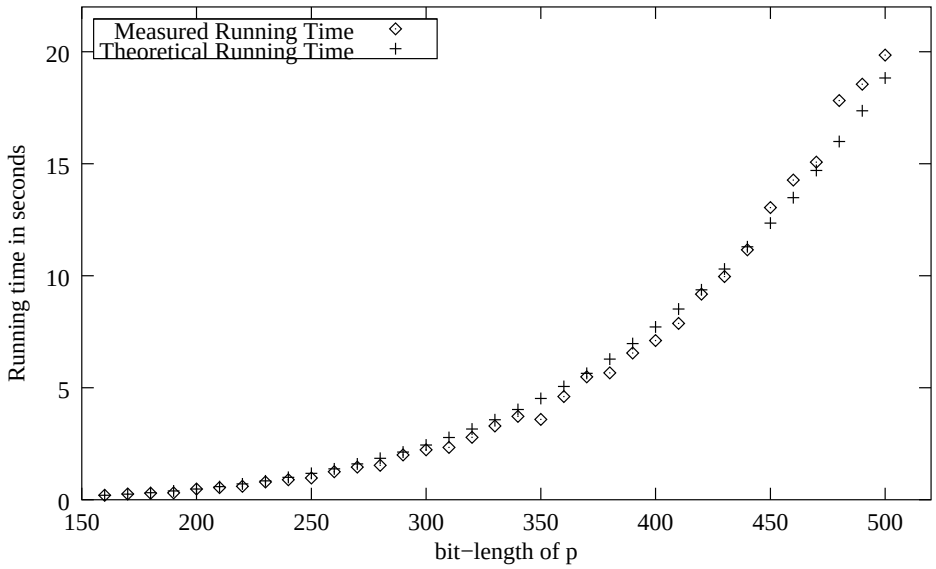


Figure 4.5: Timings on the Pentium III of `findPrime1Mod8`$(2^{b-3}, 4, -21311, 2^{b-1}, 2^b - 1)$ for $160 \leq b \leq 500$. The running time is given in seconds and plotted as a function of b. In addition, we provide a theoretical curve proportional to b^4 passing through the timing for $b = 160$.

4.3.2 Discriminants $\Delta \equiv 5 \bmod 8$

In this section we investigate discriminants $\Delta \equiv 5 \bmod 8$. We discuss in detail our optimized algorithm `findPrime5Mod8`$(r_0, 1, \Delta, [i_0, i_1])$. Its task is, for given r_0, $\Delta \equiv 5 \bmod 8$, and $[i_0, i_1]$, to find a prime p in $[i_0, i_1]$, which is suitable for the CM-method of discriminant Δ and such that there is a cryptographically strong elliptic curve of prime order $r \geq r_0$ over $\mathbb{F}_p$ with endomorphism ring $\mathcal{O}_\Delta$. The proceeding is very similar to the case $\Delta \equiv 1 \bmod 8$ of the Section 4.3.1.

As in the previous section, we first refine the conditions on a pair $(t, y) \in \mathbb{N}^2$ to properly represent a rational prime, if $\Delta \equiv 5 \bmod 8$. Hence, let a prime p be given, and let (t, y) be a

proper representation of p. In addition to the requirements of Theorem 4.2.4, as $\Delta \equiv 1 \bmod 4$, it is easy to see that both t and y are either even or odd. First, let t and y both be odd. Then $t^2 - \Delta y^2 \equiv 1 - 5 \equiv 4 \bmod 8$. Hence, $p = \frac{t^2 - \Delta y^2}{4}$, $p + 1 - t$, and $p + 1 + t$ are all odd. Next, let t and y both be even; furthermore, assume $p = \frac{t^2 - \Delta y^2}{4}$ to be odd. Then both $p + 1 - t$ and $p + 1 + t$ are even. Thus we only consider the first case and set $k_0 = 1$, that is we search for pairs $(t, y) \in \mathbb{N}^2$ such that p and either $p + 1 - t$ or $p + 1 + t$ are prime. We will show that this case actually exists.

As in the case $\Delta \equiv 1 \bmod 8$ we first collect congruence conditions on $(t \bmod m, y \bmod m)$ for small moduli m. We evaluate the moduli $m \in \{2, 3, 5, 7\}$. The congruence conditions on (t, y) for $m = 2$ are already stated. Next, for $m \in \{3, 5, 7\}$ we proceed as in the previous section. The arguments for deducing the conditions on $(t \bmod m, y \bmod m)$ are similar. Again, to avoid distinguishing different cases, we leave y unchanged. We summarize the results in Table 4.6.

m	$y \bmod m$	N_-	N_+
2	1	$t \equiv 1 \bmod 2$	$t \equiv 1 \bmod 2$
3	0	$t \equiv 1 \bmod 3$	$t \equiv 2 \bmod 3$
5	0	$t \bmod 5 \in \{1, 3, 4\}$	$t \bmod 5 \in \{1, 2, 4\}$
7	0	$t \bmod 7 \in \{1, 3, 4, 5, 6\}$	$t \bmod 7 \in \{1, 2, 3, 4, 6\}$

Table 4.6: Congruence conditions on t and y for N_- respectively N_+ to be prime in case of $\Delta \equiv 5 \bmod 8$.

Next, assume (t, y) to respect the congruence conditions of Table 4.6. We explain how to preserve the congruence conditions when modifying (t, y). First, as y is fixed, we have to ensure that t keeps odd. Hence, we successively increase t by 2 until the new residues of t modulo 3, 5 and 7 fit to the requirements of Table 4.6, that is we determine the minimal $c \in \mathbb{N}$ such that both (t, y) and $(t + 2c, y)$ respect the congruence conditions. Again, as in the case $\Delta \equiv 1 \bmod 8$, we set $t_m = t \bmod m$ for $m \in \{3, 5, 7\}$. The values of c are listed in Table 4.7.

c	(t_3, t_5, t_7)
14	(1, 3, 3), (1, 4, 4)
8	(1, 1, 4), (1, 3, 1)
5	(1, 1, 1), (1, 1, 3), (1, 1, 5), (1, 1, 6), (1, 4, 1), (1, 4, 3), (1, 4, 5), (1, 4, 6)
2	(1, 3, 4), (1, 3, 5), (1, 3, 6)
1	(2, 1, 1), (2, 1, 2), (2, 1, 3), (2, 1, 4), (2, 1, 6), (2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4), (2, 2, 6), (2, 4, 1), (2, 4, 2), (2, 4, 3), (2, 4, 4), (2, 4, 6)

Table 4.7: Minimal values c such that both $(t \bmod 3, t \bmod 5, t \bmod 7)$ and $(t + 2c \bmod 3, t + 2c \bmod 5, t + 2c \bmod 7)$ respect the congruence conditions of Table 4.6.

As in the case $\Delta \equiv 1 \bmod 8$ we introduce a method `cycles5Mod8`(t_3, t_5, t_7). Let (t_3, t_5, t_7) be an element of Table 4.7. Then `cycles5Mod8`(t_3, t_5, t_7) returns the appropriate value $c \in \mathbb{N}$ of Table 4.7. For example, `cycles5Mod8`$(1, 3, 3)$ returns 14.

Algorithm 4.7: `findPrime5Mod8`$(r_0, 1, \Delta, [i_0, i_1])$

Input: Positive integers r_0 and $k_0 = 1$.
A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 5 \bmod 8$.
An interval $[i_0, i_1] \subset \mathbb{N}$ with $i_1 - i_0 + 1 \geq 2^{159}$.
Output: The discriminant Δ.
A prime $p \in [i_0, i_1]$.
A prime $r \geq r_0$ such that there exists a cryptographically strong elliptic curve E over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = r$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
The cofactor $k = 1$.

$K \leftarrow 2^{40}$; $m \leftarrow \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor$; $T \leftarrow 2000$; $M \leftarrow 29$;
$m_R \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $s \in_R \{0, \ldots K-1\}$; $v \leftarrow i_0 + sl$; $w \leftarrow v + l - 1$; //choose a random subinterval
  $\tau \in_R \{1, ..., \lfloor 2\sqrt{v} \rfloor\}$; $\mu \leftarrow \left\lceil \sqrt{(4v - \tau^2)/|\Delta|} \right\rceil$;
  $t \leftarrow \min\{x \geq \tau : x \equiv 1 \bmod 210\}$;
  $y \leftarrow \min\{x \geq \mu : x \equiv 105 \bmod 210\}$;
  $p \leftarrow (t^2 - \Delta y^2)/4$;
  $c \leftarrow 0$; $c_r \leftarrow 0$; $j \leftarrow 0$;
  $t_3 \leftarrow 1$; $t_5 \leftarrow 1$; $t_7 \leftarrow 1$;
  for $i = 0$ to $M - 4$ **do**
    $p[i] \leftarrow p \bmod p_{i+4}$; //initialize $p[\,]$, i.e. $p \bmod 11, \ldots, p \bmod 113$
    $t[i] \leftarrow t \bmod p_{i+4}$; // initialize $t[\,]$, i.e. $t \bmod 11, \ldots, t \bmod 113$
  while $j < T$ AND $p \in [v, w]$ **do**
    if $p[i] \neq 0$ for all $0 \leq i \leq M - 4$ **then**
      $b_m \leftarrow$ `false`; $b_p \leftarrow$ `false`; // booleans to keep track of loops
      if $t_3 = 1$ AND $p[i] + 1 - t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        $p \leftarrow p + c_r t + c_r^2$; $t \leftarrow t + 2c_r$; $c_r \leftarrow 0$; $b_m \leftarrow$ `true`;
        if $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true` **then**
          $b_p \leftarrow$ `true`;
          if `isPrime`$(p + 1 - t, m_R) =$ `true` AND `isStrong`$(r_0, 1, p, p + 1 - t) \neq 0$ **then**
            return($\Delta, p, p + 1 - t, 1$);
      if $t_3 = 2$ AND $p[i] + 1 + t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        if $b_m =$ `false` **then**
          $p \leftarrow p + c_r t + c_r^2$; $t \leftarrow t + 2c_r$; $c_r \leftarrow 0$;
        if $b_p =$ `true` OR ($b_m =$ `false` AND $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true`) **then**
          if `isPrime`$(p + 1 - t, m_R) =$ `true` AND `isStrong`$(r_0, 1, p, p + 1 + t) \neq 0$ **then**
            return($\Delta, p, p + 1 + t, 1$);
    $c \leftarrow$ `cycles5Mod8`(t_3, t_5, t_7); $c_r \leftarrow c_r + c$
    $t_3 \leftarrow t_3 + 2c \bmod 3$; $t_5 \leftarrow t_5 + 2c \bmod 5$; $t_7 \leftarrow t_7 + 2c \bmod 7$;
    for $i = 0$ to $M - 4$ **do**
      $p[i] \leftarrow p[i] + ct[i] + c^2 \bmod p_{i+4}$; $t[i] \leftarrow t[i] + 2c \bmod p_{i+4}$;
    $j \leftarrow j + 1$;

We next turn to our algorithm `findPrime5Mod8`$(r_0, 1, \Delta, [i_0, i_1])$. It is very similar to algorithm `findPrime1Mod8`. Hence we refer to the latter algorithm for details. Input of `findPrime5Mod8` are positive integers r_0 and $k_0 = 1$. In addition, it requires a discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 5 \bmod 8$, and natural numbers $i_0 < i_1$. Its output is the discriminant Δ and a rational prime $p \in [i_0, i_1]$. Furthermore, it returns a rational prime $r \geq r_0$ such that there exists a cryptographically strong elliptic curve E defined over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = r$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$. Finally, `findPrime5Mod8` returns the cofactor 1.

The primes p returned by `findPrime5Mod8` are K-uniformly distributed in $[i_0, i_0 + K \cdot \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor]$, and in practice we set $K = 2^{40}$. Furthermore, the sieving arrays $p[\ ]$ and $t[\ ]$, the initialization of a pair (t, y), and the initialization parameter T are analogous to the case $\Delta \equiv 1 \bmod 8$. Our tests indicate that the choices $T = 2000$ and $M = 29$ are optimal, that is we make use of the primes in $[11, 113]$ for sieving.

We tested `findPrime5Mod8`$(r_0, 1, \Delta, [i_0, i_1])$ in practice for $r_0 = 2^{159}$, fundamental discriminants Δ of class number 200, and the interval $[2^{159}, 2^{160} - 1]$. Using this input, the running time of `findPrime5Mod8` on the Pentium III is about 0.25 - 0.3 seconds.

4.3.3 Discriminants $\Delta \equiv 0 \bmod 4$

In this section we present our algorithm `findPrime0Mod4`$(r_0, k_0, \Delta, [i_0, i_1])$. Its task is, for given r_0, k_0, $\Delta \equiv 0 \bmod 4$, and $[i_0, i_1]$, to find a prime $p \in [i_0, i_1]$, which is suitable for the CM-method of discriminant Δ and such that there is a cryptographically strong elliptic curve E over $\mathbb{F}_p$ of order rk with $r \geq r_0$, $k \leq k_0$, and endomorphism ring $\mathcal{O}_\Delta$. The proceeding is very similar to the cases $\Delta \equiv 1 \bmod 8$ and $\Delta \equiv 5 \bmod 8$ of the Sections 4.3.1 and 4.3.2, respectively. We will only consider $k_0 \in \{2; 3\}$ and $\Delta \bmod 16 \in \{8; 12\}$ as explained below.

Again we first work out congruence conditions for t and y, respectively. In this section we evaluate the moduli $m \in \{3, 4, 5, 7\}$. When we derived the congruence conditions of Table 4.6, we used the tables of Section A.1.2 of the appendix; these tables list conditions for $\Delta \not\equiv 1 \bmod 8$. Thus we make use of Table 4.6, too.

However, in Section 4.3.2 we used the modulus $m = 2$. Hence, in addition to Table 4.6 we have to deduce conditions for the modulus $m = 4$. Let p be a prime with $4p = t^2 - \Delta y^2$; modulo 4, this equation shows $t^2 \equiv 0 \bmod 4$. Thus a first condition is $2 \mid t$. We set $t_{1/2} = t/2$ and $p = t_{1/2}^2 - \frac{\Delta}{4} y^2$ in what follows. Theorem 4.2.4 shows that we have to distinguish two cases.

1. Let $\gcd(2t_{1/2}, y) = 1$. Hence y is odd. Let $\Delta \equiv 0 \bmod 8$. Thus $\frac{\Delta}{4} y^2$ is even, and $t_{1/2}$ has to be odd. Furthermore, we have $p + 1 \pm t \equiv t_{1/2}^2 - \frac{\Delta}{4} y^2 + 1 \pm 2t_{1/2} \equiv 1 - \frac{\Delta}{4} + 1 + 2 \equiv -\frac{\Delta}{4} \bmod 4$, hence

$$p + 1 \pm t \bmod 4 = \begin{cases} 0, & \text{if} \quad \Delta \equiv 0 \bmod 16\ , \\ 2, & \text{if} \quad \Delta \equiv 8 \bmod 16\ . \end{cases}$$

Thus we have $k \geq 4$ and $k \geq 2$ in case of $\Delta \equiv 0 \bmod 16$ and $\Delta \equiv 8 \bmod 16$, respectively. As we are interested in curves having a cofactor as small as possible, we assume $k_0 \in \{2; 3\}$. Hence, if $\gcd(2t_{1/2}, y) = 1$ and $\Delta \equiv 0 \bmod 8$ we only consider discriminants $\Delta \equiv 8 \bmod 16$.

Next let $\Delta \equiv 4 \bmod 8$. Then $\frac{\Delta}{4} y^2$ is odd, and $t_{1/2}$ has to be even. It follows $p + 1 \pm t \equiv -\frac{\Delta}{4} y^2 + 1 \equiv -\frac{\Delta}{4} + 1 \bmod 4$, hence

$$p+1 \pm t \bmod 4 = \begin{cases} -1+1 &= 0, \quad \text{if} \quad \Delta \equiv 4 \bmod 16\,, \\ -3+1 &= 2, \quad \text{if} \quad \Delta \equiv 12 \bmod 16\,. \end{cases}$$

Thus we have $k \geq 4$ and $k \geq 2$ in case of $\Delta \equiv 4 \bmod 16$ and $\Delta \equiv 12 \bmod 16$, respectively. Thus, if $\gcd(2t_{1/2}, y) = 1$ and $\Delta \equiv 4 \bmod 8$, we only consider discriminants $\Delta \equiv 12 \bmod 16$.

2. Let $\gcd(2t_{1/2}, y) = 2$ with odd $t_{1/2}$. We write $y = 2y_{1/2}$. Then we have $p = t_{1/2}^2 - \Delta y_{1/2}^2 \equiv 1 \bmod 4$, and $p+1 \pm t \equiv 1+1+2 \equiv 0 \bmod 4$ follows. Hence we have $k \geq 4$ in this case, and we ignore pairs (t, y) with $\gcd(t, y) = 2$.

We abstain from a further investigation of discriminants $\Delta \equiv 0, 4 \bmod 16$ as in either case the discriminant is not fundamental and the cofactor at least 4. For discriminants $\Delta \equiv 8, 12 \bmod 16$ we summarize the results in Table 4.8.

$\Delta \bmod 16$	k	$(t \bmod 4, y \bmod 4)$
8	2	(2,1), (2,3)
12	2	(0,1), (0,3)

Table 4.8: Pairs $(t \bmod 4, y \bmod 4)$ such that p and either $\frac{N_-}{k}$ or $\frac{N_-}{k}$ is odd with minimal k in case of $\Delta \equiv 0 \bmod 4$.

We next explain how to initialize and modify a pair (t, y). Again, in order to avoid distinguishing different cases, we fix y with $y \bmod 2 = 1$. Thus, if $\Delta \equiv 8 \bmod 16$ we have to ensure $t \bmod 4 = 2$, and $t \bmod 4 = 0$ otherwise. Furthermore, we have to respect the requirements of Table 4.6. Hence, in addition, we initialize t and y such that $t \equiv 1 \bmod 105$ and $y \equiv 0 \bmod 105$, respectively. However, as $t \bmod 4$ has to rest constant, we increase t by 4 in each step. Now, we assume that (t, y) respects all requirements of the Tables 4.6 and 4.8. As in the previous sections, we list in Table 4.9 minimal values c such that $(t+4c, y)$ respects the congruence conditions, too. This value is returned by our algorithm `cycles0Mod4`. Again, we set $t_m = t \bmod m$ for $m \in \{3, 5, 7\}$.

c	(t_3, t_5, t_7)
8	(2,1,1), (2,2,2)
7	(1,3,3), (1,4,4)
5	(2,1,6), (2,4,2)
4	(1,1,4), (1,3,1)
3	(1,1,1), (1,1,3), (1,1,5), (1,1,6), (1,4,1), (1,4,3), (1,4,5), (1,4,6), (2,2,1), (2,2,3), (2,2,4), (2,2,6), (2,4,1), (2,4,3), (2,4,4), (2,4,6)
2	(2,1,2), (2,1,3), (2,1,4)
1	(1,3,4), (1,3,5), (1,3,6)

Table 4.9: Minimal values c such that both $(t \bmod 3, t \bmod 5, t \bmod 7)$ and $(t+4c \bmod 3, t+4c \bmod 5, t+4c \bmod 7)$ respect the congruence conditions of Tables 4.6 and 4.8. We assume $\Delta \equiv 0 \bmod 4$.

We next present our efficient algorithm `findPrime0Mod4`. After the detailed discussion of the

previous algorithms `findPrime1Mod8` and `findPrime5Mod8`, the design details are obvious.

Finally, we present practical running times of `findPrime0Mod4`. We tested our implementation on the Pentium III for $r_0 = 2^{179}$, $k_0 = 2$, and $[i_0, i_1] = [2^{180}, 2^{181} - 1]$, that is the bitlength of r is 180. For $200 \leq h \leq 6000$, $200 \mid h$ we determined the maximal discriminants $\Delta \equiv 8 \bmod 16$ and $\Delta \equiv 12 \bmod 16$, respectively, with $|\Delta| < 2.5 \cdot 10^7$, if such a discriminant exists. For each input we performed 250 tests. The timings are shown in Figure 4.6. We are not able to determine the correlation of timing and class number.

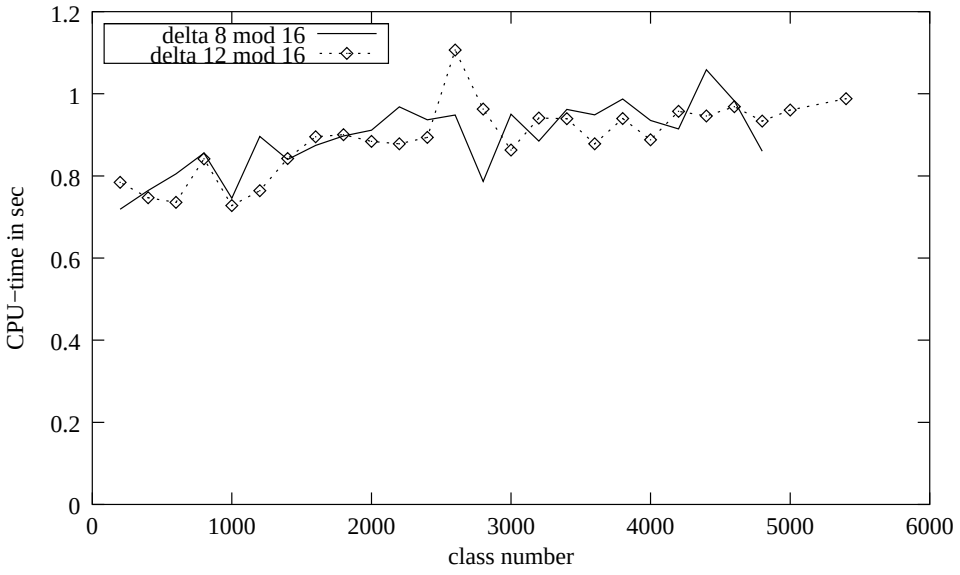


Figure 4.6: Timings on the Pentium III of `findPrime0Mod4`$(2^{179}, 2, \Delta, 2^{180}, 2^{181} - 1)$ for maximal discriminants $\Delta \equiv 8 \bmod 16$ and $\Delta \equiv 12 \bmod 16$, respectively, of class numbers $200 \leq h \leq 6000$, $200 \mid h$. The running time is given in seconds and plotted as a function of h.

Algorithm 4.8: `findPrime0Mod4`$(r_0, k_0\Delta, [i_0, i_1])$

Input: Positive integers r_0 and k_0.
A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 8, 12 \bmod 16$.
An interval $[i_0, i_1] \subset \mathbb{N}$ with $i_1 - i_0 + 1 \geq 2^{159}$.
Output: The discriminant Δ.
A prime $p \in [i_0, i_1]$.
A prime $r \geq r_0$ and an integer $k \leq k_0$ such that a cryptographically strong elliptic curve E over $\mathbb{F}_p$ exists with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.

$K \leftarrow 2^{40}$; $m \leftarrow \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor$; $T \leftarrow 2000$; $M \leftarrow 24$;
if $\Delta \equiv 8 \bmod 16$ **then**
  $t_4 \leftarrow 2$; //if $\Delta \equiv 8 \bmod 16$, ensure $t \equiv 2 \bmod 4$
else
  $t_4 \leftarrow 0$; //if $\Delta \equiv 12 \bmod 16$, ensure $t \equiv 0 \bmod 4$
$m_R \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $s \in_R \{0, \ldots K-1\}$; $v \leftarrow i_0 + sl$; $w \leftarrow v + l - 1$; //choose a random subinterval
  $\tau \in_R \{1, ..., \lfloor 2\sqrt{v} \rfloor\}$; $\mu \leftarrow \left\lceil \sqrt{(4v - \tau^2)/|\Delta|} \right\rceil$;
  $t \leftarrow \min\{x \geq \tau : x \equiv 1 \bmod 105, x \equiv t_4 \bmod 4\}$;
  $y \leftarrow \min\{x \geq \mu : x \equiv 105 \bmod 210\}$;
  $p \leftarrow (t^2 - \Delta y^2)/4$;
  $c \leftarrow 0$; $c_r \leftarrow 0$; $j \leftarrow 0$;
  $t_3 \leftarrow 1$; $t_5 \leftarrow 1$; $t_7 \leftarrow 1$;
  for $i = 0$ to $M - 4$ **do**
    $p[i] \leftarrow p \bmod p_{i+4}$; //initialize $p[\,]$, i.e. $p \bmod 11, \ldots, p \bmod 97$
    $t[i] \leftarrow t \bmod p_{i+4}$; // initialize $t[\,]$, i.e. $t \bmod 11, \ldots, t \bmod 97$
  while $j < T$ AND $p \in [v, w]$ **do**
    if $p[i] \neq 0$ for all $0 \leq i \leq M - 4$ **then**
      $b_m \leftarrow$ `false`; $b_p \leftarrow$ `false`; // booleans to keep track of loops
      if $t_3 = 1$ AND $p[i] + 1 - t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        $p \leftarrow p + 2c_r t + 4c_r^2$; $t \leftarrow t + 4c_r$; $c_r \leftarrow 0$; $b_m \leftarrow$ `true`;
        if $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true` **then**
          $b_p \leftarrow$ `true`;
          if `isPrime`$(p + 1 - t, m_R) =$ `true` AND $(r \leftarrow$ `isStrong`$(r_0, k_0, p, p + 1 - t)) \neq 0$ **then**
            return($\Delta, p, r, (p + 1 - t)/r$);
      if $t_3 = 2$ AND $p[i] + 1 + t[i] \bmod p_{i+4} \neq 0$ for all $0 \leq i \leq M - 4$ **then**
        if $b_m =$ `false` **then**
          $p \leftarrow p + 2c_r t + 4c_r^2$; $t \leftarrow t + 4c_r$; $c_r \leftarrow 0$;
        if $b_p =$ `true` OR ($b_m =$ `false` AND $p \in [v, w]$ AND `isPrime`$(p, m_R) =$ `true`)) **then**
          if `isPrime`$(p + 1 - t, m_R) =$ `true` AND $(r \leftarrow$ `isStrong`$(r_0, k_0, p, p + 1 + t)) \neq 0$ **then**
            return($\Delta, p, r, (p + 1 + t)/r$);
    $c \leftarrow$ `cycles0Mod4`(t_3, t_5, t_7); $c_r \leftarrow c_r + c$
    $t_3 \leftarrow t_3 + 2c \bmod 3$; $t_5 \leftarrow t_5 + 2c \bmod 5$; $t_7 \leftarrow t_7 + 2c \bmod 7$;
    for $i = 0$ to $M - 4$ **do**
      $p[i] \leftarrow p[i] + 2ct[i] + 4c^2 \bmod p_{i+4}$; $t[i] \leftarrow t[i] + 4c \bmod p_{i+4}$;
    $j \leftarrow j + 1$;

Chapter 5

Finding a Suitable Cardinality: A Fixed Field Approach

In this chapter we present our algorithm `findDiscriminant`(p, r_0, k_0, h_0). Its task is as follows: Let a rational prime p and the security bounds r_0, k_0, and h_0 be given. The algorithm finds a fundamental discriminant Δ with $h(\Delta) \geq h_0$ such that p is suitable for the CM-method of discriminant Δ. In addition, if $p = \frac{t^2 - \Delta y^2}{4}$, we have either $p + 1 - t = rk$ or $p + 1 + t = rk$ with a rational prime $r \geq r_0$ and a positive integer $k \leq k_0$. Finally, rk is the order of a cryptographically strong elliptic curve E over $\mathbb{F}_p$ with $\mathrm{End}(E) = \mathcal{O}_\Delta$. Throughout this chapter we assume that the bitlength of p is reasonable, that is $\lfloor \log_2 p \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$.

Our algorithm `findDiscriminant` bases on ideas due to Atkin and Morain ([AM93]) and the abstractly formulated algorithm in section E.3.2.c of [X9.62]. However, [AM93] describe their algorithm in the context of primality proving; hence they do not supply an algorithm for our purposes. Furthermore, the algorithm in [X9.62] uses a high-level description and does not address the problem of how to choose appropriate candidates of discriminants. We will show how to extend both approaches to find an elliptic curve for use in cryptography. In addition, we show how to efficiently implement our algorithm. Furthermore, in contrast to [X9.62], we explicitely integrate the search for the discriminant in our algorithm, and we take the parameter h_0 into account. Finally, we give evidence that for $r_0 \geq 2^{159}$, $k_0 = 1$, and $h_0 = 200$, our generating algorithm `cryptoCurve` is rather fast in practice if the prime p is set in algorithm `findPrime`. For instance, if $r_0 = 2^{159}$, we expect `cryptoCurve` to terminate successfully in about a minute on the Pentium III.

Before explaining the algorithm, we first discuss some underlying details. We assume Δ to be fundamental. Thus Δ factors as a product of pairwise different odd primes and a further factor d, where $d \in \{-1, -4, -8\}$. Again, we denote by p_j the j-th odd prime, and we write

$$\Delta = -2^e \cdot \prod_{j=0}^{M-1} \delta_j p_{j+1} \cdot \prod_{k=1}^{L} q_k \,, \tag{5.1}$$

with $e \in \{0, 2, 3\}$. The further variables have the following meaning: M is the index of the maximal odd prime which we use to speed up the decision whether p is suitable for the CM-method of the current discriminant Δ. In our implementation, we set $M = 25$, that is we consider the primes in $[3, 101]$. Furthermore, for $0 \leq j \leq M - 1$ we have $\delta_j = 1$ if and only if

$p_{j+1} \mid \Delta$, and $\delta_j = 0$ otherwise. Finally, the factors q_k denote all odd primes $> p_M$ dividing Δ. In addition, as in [X9.62], we set $D = -\Delta$, if Δ is odd, and $D = -\Delta/4$, if Δ is even. Hence, D is equal to the square-free part of $-\Delta$. We make use of the following congruence conditions quoted in [X9.62], section E.3.2.1., p. 113:

$$\begin{aligned} p \equiv 3 \bmod 8 &\implies D \not\equiv 6 \bmod 8\ , \\ p \equiv 5 \bmod 8 &\implies D \equiv 1 \bmod 2\ , \\ p \equiv 7 \bmod 8 &\implies D \not\equiv 2 \bmod 8\ . \end{aligned}$$

We next derive further conditions on Δ basing on Legendre symbols. These ideas are already mentioned in [AM93]. We assume, that p is suitable for the CM-method of discriminant Δ. If l is an odd prime dividing Δ, the representation $4p = t^2 - \Delta y^2$ becomes modulo l

$$p \equiv (t/2)^2 \bmod l\ .$$

Hence, if p is suitable for the CM-method of discriminant Δ, $\left(\frac{p}{l}\right) = 1$ follows for all odd prime divisors of Δ.

Now let Δ be a fundamental imaginary quadratic discriminant. We assume that the factorization (5.1) is known. In addition, let p be an odd rational prime. In order to efficiently check the above conditions we introduce integers B_p and B_Δ defined as follows:

$$B_p = \sum_{j=0}^{M+2} \theta_j \cdot 2^j, \qquad \theta_j = \begin{cases} 1, & \text{if} \quad \left(\frac{p}{p_{j+1}}\right) \neq 1, \quad 0 \leq j \leq M-1, \\ 1, & \text{if} \quad p \equiv 3 \bmod 8, \quad j = M, \\ 1, & \text{if} \quad p \equiv 5 \bmod 8, \quad j = M+1, \\ 1, & \text{if} \quad p \equiv 7 \bmod 8, \quad j = M+2, \end{cases}$$

$$B_\Delta = \sum_{j=0}^{M+2} \delta_j \cdot 2^j, \qquad \delta_j = \begin{cases} 1, & \text{if} \quad D \equiv 6 \bmod 8, \quad j = M, \\ 1, & \text{if} \quad D \equiv 0 \bmod 2, \quad j = M+1, \\ 1, & \text{if} \quad D \equiv 2 \bmod 8, \quad j = M+2, \end{cases}$$

and $\theta_j = 0$ respectively $\delta_j = 0$ otherwise for either j. Thus, in order for p to be suitable for the CM-method of discriminant Δ, a necessary condition is $B_p \,\&\, B_\Delta = 0$, where & denotes the bitwise AND-operation. Given B_p and B_Δ, we can check this condition very fast in practice. Finally, as deduced in Section 2.1, a further necessary condition for p to be suitable for the CM-method of discriminant Δ is $\left(\frac{\Delta}{p}\right) = 1$. We show that for a discriminant of odd class number, this condition implies $\left(\frac{p}{l}\right) = 1$ for all odd prime divisors l of Δ. We first remark that if $h(\Delta)$ is odd, $-\Delta$ is prime. Now assume $\left(\frac{\Delta}{p}\right) = 1$. We have to show $\left(\frac{p}{-\Delta}\right) = 1$. We may assume $2 \neq p \neq -\Delta$ in our application. Thus, using quadratic reciprocity, we have $\left(\frac{p}{-\Delta}\right) \cdot \left(\frac{-\Delta}{p}\right) = (-1)^{(p-1)(-\Delta-1)/4}$. We have $\left(\frac{-\Delta}{p}\right) = \left(\frac{-1}{p}\right) \cdot \left(\frac{\Delta}{p}\right) = \left(\frac{-1}{p}\right)$, and in addition $\left(\frac{-1}{p}\right) = 1$ and $\left(\frac{-1}{p}\right) = -1$, if $p \equiv 1 \bmod 4$ and $p \equiv 3 \bmod 4$, respectively. Furthermore, as $-\Delta$ is prime, we have $\Delta \equiv 1 \bmod 4$; this shows $-\Delta \equiv 3 \bmod 4$, thus $(-1)^{(p-1)(-\Delta-1)/4} = (-1)^{(p-1)/2}$. Hence, we have $\left(\frac{p}{-\Delta}\right) = (-1)^{(p-1)/2} \cdot \left(\frac{-1}{p}\right)$. If either $p \equiv 1 \bmod 4$ or $p \equiv 3 \bmod 4$, this proves the assertion.

We next explain how to choose suitable candidates of discriminants. Basing on our database delta_h explained on Page 45, we determined all fundamental discriminants Δ with $|\Delta| < 6{\cdot}10^6$

and $200 \leq h(\Delta) \leq 999$. We store these discriminants with respect to their class number in a further database, which we call fundamental_delta_h. In addition, if the class number is even, we store B_Δ, L and Q together with Δ, where Q denotes an array of length L which contains the prime factors q_k, that is $Q[k-1] = q_k$ for all $1 \leq k \leq L$.

In order to be able to use our database fundamental_delta_h, we introduce the function `getDiscriminant`(h, Δ_0) : Its input is a positive integer h with $200 \leq h \leq 999$ and an integer Δ_0 with $|\Delta_0| < 6 \cdot 10^6$. Using the database fundamental_delta_h, `getDiscriminant`(h, Δ_0) returns the maximal imaginary quadratic fundamental discriminant Δ with $h(\Delta) = h$, $\Delta \leq \Delta_0$, and $|\Delta_0| < 6 \cdot 10^6$ if such a discriminant exists. Otherwise, `getDiscriminant` returns 0. In addition, if h is even, the function returns B_Δ, L, and the array Q.

We are now able to explain our algorithm `findDiscriminant`. Its input are the prime p and the bounds r_0, k_0, and h_0. In order to make use of our database fundamental_delta_h, we assume $200 \leq h_0 \leq 999$. `findDiscriminant` first computes B_p. In order to get a first candidate of discriminant, the algorithm then invokes `getDiscriminant`$(h_0, 0)$. Let Δ denote the discriminant returned by `getDiscriminant`. If h_0 is odd, we compute $\left(\frac{\Delta}{p}\right)$. If the Legendre symbol is not equal to 1, we turn to the next discriminant by invoking `getDiscriminant`(h_0, Δ). Otherwise, we make use of the algorithm of Cornacchia to decide whether p is suitable for the CM-method of discriminant Δ; the according function `cornacchia`(Δ, p) was already discussed in Section 4.1.1. If this is not the case, we choose the next discriminant. Otherwise, using the representation $4p = t^2 - \Delta y^2$, we check if either $p+1-t$ or $p+1+t$ is the order of a cryptographically strong elliptic curve. If this is false, we invoke `getDiscriminant`(h_0, Δ). Otherwise, the algorithm returns the values Δ, p, r, and k. If h_0 is even, we first compute $B_p \,\&\, B_\Delta$. If the result is a positive integer, we turn to the next discriminant. Otherwise, for $1 \leq k \leq L$ we successively compute $\left(\frac{p}{q_k}\right)$. If one of the Legendre symbols is not equal to 1, we choose the next discriminant. Otherwise, we compute $\left(\frac{\Delta}{p}\right)$ and proceed as in the case of odd h_0. If we are not successful until `getDiscriminant`(h_0, Δ) returns 0, we set $h = h_0 + 1$ and invoke `getDiscriminant`$(h, 0)$. We then proceed as in the case h_0. Finally, if we are not able to find suitable values Δ, r and k before $h = 1000$, the algorithm outputs an error message and terminates. However, in practice, we set $h_0 = 200$, $2^{159} \leq r_0 \leq 2^{249}$, and $1 \leq k_0 \leq 4$. We are not aware of any prime p where our algorithm fails.

We demonstrate the efficiency of `findDiscriminant` in practice. In Section A.2 we present sample output of algorithm `findDiscriminant`(p, r_0, k_0, h_0) for different bitlengths of p. If p is a prime of bitlength b, we used $r_0 = 2^{b-1}$, $k_0 = 1$, and $h_0 = 200$ as input of our algorithm `findDiscriminant`. We tested our algorithm for $b = 160$ and $b = 250$, that is r is of bitlength 160 and 250, respectively. For each b, we performed 100 tests. We first generated 100 random primes of bitlength 160 and 250, respectively. For each input `findDiscriminant` terminated successfully. All tests were performed on the Pentium III.

Let us first consider the case $b = 160$. The average running time of `findDiscriminant` was 23.6 seconds. In addition the average timing of the whole generating algorithm `cryptoCurve` was 65.6 seconds. The best running time of `cryptoCurve` was 26.4 seconds; the corresponding discriminant was -988867 of class number 200. In contrast, the discriminant of maximal class number was $\Delta = -4198963$ with $h(\Delta) = 319$. The corresponding running time of `findDiscriminant` and `cryptoCurve` was 140 seconds and about 259 seconds, respectively. We remark that for our sample test primes p the timings of `findDiscriminant` are 72.6%

Algorithm 5.1: `findDiscriminant`(p, r_0, k_0, h_0)

Input: A prime p and positive integers r_0, k_0 with $\lfloor \log_2 p \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$.
An integer h_0 with $200 \leq h_0 \leq 999$.

Output: A fundamental imaginary quadratic discriminant Δ with $|\Delta| < 6 \cdot 10^6$ and $999 \geq h(\Delta) \geq h_0$.
The prime p.
A rational prime r and a positive integer k with $r \geq r_0$, $k \leq k_0$, and $\lfloor \log_2 rk \rfloor = \lfloor \log_2 r_0 k_0 \rfloor$, such that a cryptographically strong elliptic curve E over $\mathbb{F}_p$ exists with $|E(\mathbb{F}_p)| = rk$ and $\mathrm{End}(E) = \mathcal{O}_\Delta$.
An error message, if no appropriate values Δ, r and k exist.

```
Compute B_p; h ← h_0; Δ ← 0;
while h ≤ 999 do
  if h ≡ 0 mod 2 then
    (Δ, B_Δ, L, Q) ← getDiscriminant(h, Δ);
    if Δ = 0 then
      h ← h + 1; continue;
    if B_p & B_Δ > 0 OR (p/Q[j−1]) ≠ 1 for some j with 1 ≤ j ≤ L then
      continue;
  else
    Δ ← getDiscriminant(h, Δ);
    if Δ = 0 then
      h ← h + 1; continue;
  if (Δ/p) ≠ 1 OR (t ← cornacchia(Δ, p)) = 0 then
    continue;
  if (r ← isStrong(r_0, k_0, p, p + 1 − t)) ≠ 0 then
    return( Δ, p, r, (p + 1 − t)/r );
  else if (r ← isStrong(r_0, k_0, p, p + 1 + t)) ≠ 0 then
    return( Δ, p, r, (p + 1 + t)/r );
Output( "We are not able to find a suitable cardinality." ); terminate;
```

faster than a variant of `findDiscriminant` without using Legendre symbols, which took us 86.8 seconds on average.

Next, we turn to sample tests for $b = 250$. In this case, the average running time of `findDiscriminant` was 104 seconds. Furthermore, the average timing of the whole generating algorithm `cryptoCurve` was 176 seconds. The fastest timing of `cryptoCurve` was 42.5 seconds; the corresponding discriminant was -450059 of class number 200. In contrast, the discriminant of maximal class number was $\Delta = -4797227$ with $h(\Delta) = 519$. Its run time of `findDiscriminant` and `cryptoCurve` was 805 seconds and about 21 minutes, respectively.

We conclude that the Fixed Field Approach turns out to be faster than using randomly chosen curves. This statement becomes more evident with increasing bitlength of p. However, it turns out that this approach is slower in practice than the Fixed Discriminant Approach. The main reason is that for given h_0 `findDiscriminant` returns in general a discriminant Δ with $h(\Delta) > h_0$, which increases the running time.

Chapter 6

Computation of the Class Group

In this chapter we investigate two different algorithms to compute all reduced representatives of integral binary quadratic forms of discriminant Δ. It turns out that a brute-force like algorithm is the fastest way in practice to solve this task. We call this well known algorithm `classGroup`; a variant of it may be found in [Coh95], algorithm 5.3.5, p. 228. Our second algorithm called `classGroupDivisors` refines `classGroup`; it replaces the brute-force approach by considering necessary conditions for reduced representatives, that is it only takes divisors of a given integer into account. However, it turns out that `classGroupDivisors` is rather inefficient in practice. We present running times of both algorithms at the end of this chapter. As sample input we use discriminants of order of magnitude -10^6. While the running time of `classGroup` is about 0.2 seconds on the SUN UltraSPARC-IIi, `classGroupDivisors` is about a factor 26 slower in practice.

We first recall some facts concerning class groups from Section 2.1. If an imaginary quadratic discriminant Δ is given, a reduced representative of an integral binary quadratic form of discriminant Δ is a triple $(a, b, c) \in \mathbb{Z}^3$, where a and c are positive integers, $\gcd(a, b, c) = 1$, $|b| \leq a \leq c$, and finally $b > 0$ if either $|b| = a$ or $a = c$. The relation between Δ and a reduced representative (a, b, c) is given by $\Delta = b^2 - 4ac$. Considering this equation modulo 2, we deduce $b \equiv \Delta \bmod 2$. We next state and prove some useful conditions on a, b, and c for (a, b, c) to be a reduced representative.

Proposition 6.1.1 *Let Δ be an imaginary quadratic discriminant, and let (a, b, c) be a reduced representative of discriminant Δ. Furthermore, let $B = \lfloor\sqrt{|\Delta|/3}\rfloor$. Then $a \leq B$ and $a^2 \leq (b^2 - \Delta)/4$.*

Proof: We prove the first assertion $a \leq B$. Hence let (a, b, c) be a reduced representative of discriminant Δ. Then we have $|b| \leq a \leq \frac{b^2-\Delta}{4a}$. As a is positive, we deduce $4a^2 \leq b^2 - \Delta \leq a^2 - \Delta$. This proves the first assertion. The second assertion is an obvious consequence of the property $a \leq c = (b^2 - \Delta)/(4a)$ of a reduced representative. □

In our algorithms `classGroup` and `classGroupDivisors` we make use of $(b^2 - \Delta)/4$ as an upper bound of a^2. As in Proposition 6.1.1 we set $B = \lfloor\sqrt{|\Delta|/3}\rfloor$ in what follows.

We next explain our algorithm `classGroup`(Δ), which is rather trivial. It requires an imaginary quadratic discriminant Δ and returns an array R storing the reduced representatives

(a, b, c) of discriminant Δ. Furthermore, it returns the class number $h(\Delta)$. `classGroup` first initializes $b = \Delta \bmod 2$, computes the bound B, and sets $q = (b^2 - \Delta)/4$; hence $(1, b, q)$ is the main form of discriminant Δ. In addition, the algorithm initializes the variable h holding the current length of R; thus, when the algorithm terminates successfully, h is equal to the class number. Next, we determine all reduced representatives of the form $(a, \Delta \bmod 2, c)$. Hence, we leave b unchanged, and for all a, $2 \leq a \leq \sqrt{q}$, we test whether $a \mid q$ is true. If this is not the case, we increase a by 1 and, if $a^2 \leq q$ is also true, test for the adapted value of a whether $a \mid q$ holds. Otherwise we check if $\gcd(a, b, q/a) = 1$. If this turns out to be false, we turn to the next value a. Otherwise, $(a, b, q/a)$ is a reduced representative; thus, we store it in R and increase the counting variable h. We next decide whether $(a, -b, q/a)$ is a further reduced representative. We only have to check the conditions $0 \neq b$, $b < a$, and $a < q/a$ for the reduced representative $(a, b, q/a)$. If all conditions hold, we also store $(a, -b, q/a)$ in R and again increase h. We then turn to the next value of a.

We increase a by 1 until $a > \sqrt{q}$. We then adapt b and q as follows: By b' and q' we denote the adapted values of b and q, respectively. As we have to preserve the parity of b, we set $b' = b + 2$. If $b' > B$ the algorithm returns the pair (R, h) and terminates. Otherwise we set $q' = (b'^2 - \Delta)/4 = q + b + 1$. Finally, we identify b with b' and q with q'. As described above we proceed for all values a with $b \leq a \leq \sqrt{q}$. In Proposition 6.1.2, we prove that the bit-complexity of `classGroup` is at most $O(h^2 \log^3 h)$.

Algorithm 6.1: `classGroup`(Δ)

Input: An imaginary quadratic discriminant Δ.

Output: An array R storing the reduced representatives of integral binary quadratic forms of discriminant Δ.
The class number $h(\Delta)$.

$b \leftarrow \Delta \bmod 2$; $B \leftarrow \lfloor \sqrt{|\Delta|/3} \rfloor$; $q \leftarrow (b^2 - \Delta)/4$;
$R[0] \leftarrow (1, b, q)$; $a \leftarrow 2$; $h \leftarrow 1$; //h stores the current length of R
while $b \leq B$ **do**
  while $a \leq \sqrt{q}$ **do**
    if $a \mid q$ AND $\gcd(a, b, q/a) = 1$ **then**
      $R[h] \leftarrow (a, b, q/a)$; $h \leftarrow h + 1$;
      if $b \neq 0$ AND $b < a$ AND $a < q/a$ **then**
        $R[h] \leftarrow (a, -b, q/a)$; $h \leftarrow h + 1$;
    $a \leftarrow a + 1$;
  $q \leftarrow q + b + 1$; $b \leftarrow b + 2$; $a \leftarrow b$;
return (R, h);

Proposition 6.1.2 *Let Δ be an imaginary quadratic discriminant of class number h. The bit-complexity of* `classGroup`(Δ) *is at most $O(h^2 \log^3 h)$.*

Proof: We may neglect the complexity of computing the initial values of B and q, respectively. In all we pass $O(\sqrt{|\Delta|})$ times through the outer while-loop. Obviously $q = (b^2 - \Delta)/4 \leq (-\Delta/3 - \Delta)/4 = |\Delta|/3$. Hence, for each b we pass at most $O(\sqrt{|\Delta|})$ times through the inner while-loop. Thus in all we have to compute $O(|\Delta|)$ greatest common divisors using

the Euclidian algorithm of complexity at most $O(\log^3|\Delta|)$ (see e.g. [Coh95], p. 13). As the division q/a is of complexity $O(\log^2|\Delta|)$, we may neglect this step. In addition, we do not have to take the adaptation of q, a, and b into account. Hence, in all, the complexity of `classGroup`(Δ) is $O(|\Delta|\log^3|\Delta|)$. Finally, using the asymptotic formula $|\Delta| = O(h^2)$ of Theorem 2.1.17, shows the assertion. □

In order to develop our algorithm `classGroupDivisors`, we refine algorithm `classGroup` as follows. The main idea is to replace the brute-force proceeding with respect to a by only taking divisors of the current value $(b^2-\Delta)/4$ into account. More precisely, for fixed b we determine all divisors a of $q = (b^2-\Delta)/4$ with $b \le a \le \sqrt{q}$. This task is done by our algorithm `findDivisors`(b,q) explained below. For each divisor a in this interval we test whether $(a,b,q/a)$ is a reduced representative or not.

We explain algorithm `findDivisors`(b,q). Input of `findDivisors` are non-negative integers b and q. The algorithm returns an array D storing all divisors of q in the interval $[b,\sqrt{q}]$. In addition, it returns the length of D. The proceeding of `findDivisors`(b,q) is quite simple. It first computes the prime factorization of q; let $\prod_{i=1}^{L} p_i^{e_i}$ denote this factorization. For all vectors $(d_1,\ldots,d_L)$, $0 \le d_i \le e_i$, it tests whether $\prod_{i=1}^{L} p_i^{d_i} \in [b,\sqrt{q}]$. If this is true, $\prod_{i=1}^{L} p_i^{d_i}$ is stored in D. Otherwise, `findDivisors` turns to the next divisor of q.

Algorithm 6.2: `classGroupDivisors`(Δ)

Input: An imaginary quadratic discriminant Δ.

Output: An array R storing the reduced representatives of integral binary quadratic forms of discriminant Δ.
The class number $h(\Delta)$.

$b \leftarrow \Delta \bmod 2$; $B \leftarrow \lfloor\sqrt{|\Delta|/3}\rfloor$; $q \leftarrow (b^2-\Delta)/4$;
$R[0] \leftarrow (1,b,q)$; $a \leftarrow 2$; $h \leftarrow 1$; //h stores the current length of R
while $b \le B$ **do**
  $(D,l) \leftarrow$ `findDivisors`(b,q); $k \leftarrow 1$;
  while $k \le l$ **do**
    $a \leftarrow D[k-1]$; $k \leftarrow k+1$;
    if $\gcd(a,b,q/a) = 1$ **then**
      $R[h] \leftarrow (a,b,q/a)$; $h \leftarrow h+1$;
      if $b \ne 0$ AND $b < a$ AND $a < q/a$ **then**
        $R[h] \leftarrow (a,-b,q/a)$; $h \leftarrow h+1$;
  $q \leftarrow q+b+1$; $b \leftarrow b+2$;
return (R,h);

In order to determine the complexity of `classGroupDivisors`, we have to estimate the complexity of computing the prime factorization. However, we make use of the according LiDIA-implementation; this implementation mixes different approaches: the trial division, the Quadratic Sieve, and the Elliptic Curve Method. Hence, it is not possible to get a closed formula of the complexity of `classGroupDivisors`.

Instead, we compare practical running times of our algorithms. For all imaginary quadratic discriminants Δ, $-1.000.000 \ge \Delta \ge -2.000.000$, we determine the practical running time of

our algorithms `classGroup`(Δ) and `classGroupDivisors`(Δ), respectively. All timings are measured on the SUN UltraSPARC-IIi. Sample timings are given in Table 6.1. We deduce that for discriminants in the given interval, `classGroupDivisors` is about 26 times slower than `classGroup`. In addition, we see that the practical running time seems to depend on the range of $|\Delta|$ rather than on h.

Δ	$h(\Delta)$	time t_1 of `classGroup`	time t_2 of `classGroupDivisors`	t_2/t_1
-1000000	200	0.18	4.70	26.1
-1025000	400	0.19	5.49	28.9
-1050000	400	0.19	5.27	27.7
-1075000	300	0.19	5.40	28.4
-1100000	600	0.21	5.56	26.5

Table 6.1: Running times of our algorithms to compute the class group of discriminant Δ. All timings are given in seconds and are measured on the SUN UltraSPARC-IIi.

Chapter 7

Efficient Computation of Class Invariants

In this chapter we deal with the problem of how to efficiently compute the roots of a class polynomial. The result of our investigation is our algorithm `computeClassInvariants`(Δ, h, R), which we will present in Section 7.3. As input the algorithm requires an imaginary quadratic discriminant Δ, its class number h, and an array R storing the h reduced representatives of discriminant Δ. `computeClassInvariants` returns an array I holding the h roots of a monic polynomial with integer coefficients, which generates the ring class field $L/\mathbb{Q}(\sqrt{\Delta})$. Hence I stores the roots of a *class polynomial*. Depending on $\Delta \bmod 24$, `computeClassInvariants` computes the roots of the polynomial H, G, or W, respectively; we refer to Section 2.2.3 for details.

The underlying number-theoretical functions are the modular function j, its cube root γ_2, the Weber functions $\mathfrak{f}$, $\mathfrak{f}_1$ and $\mathfrak{f}_2$, and finally Dedekind's $\eta-$function. Let g denote one of these functions, and let (a, b, c) be a reduced representative stored in R. In Section 2.1 we identified (a, b, c) with the complex number $\tau = (-b + i\sqrt{|\Delta|})/(2a)$. The main routine of `computeClassInvariants` is to evaluate $g(\tau)$. Hence, we investigate different algorithms for this task in Section 7.2. However, our implementation computes $g(\tau)$ within a fixed floating point precision F, that is we compute a complex number g_τ such that $|g_\tau - g(\tau)| < 10^{-F}$, and we equate $g(\tau)$ and g_τ. We will explain in Chapter 9 how to choose F for a class polynomial H, G, and W, respectively. Our algorithm `getPrecision`(Δ), discussed in Chapter 9, returns the according precision F.

In order to measure the efficiency of our algorithms we compare the running times of different representations of either function in practice. It is easy to see that each function may be represented as a Fourier series on $\mathfrak{h}$. Thus we first show in Section 7.1 how to efficiently determine the Fourier coefficients of the functions involved. Next, in Section 7.2 we compare different representations of either function g to evaluate $g(\tau)$ in practice. As far as Dedekind's η-function is concerned we will show that using a representation as a sum, which is due to L. Euler, is most efficient in practice.

Furthermore, it turns out that using the η-function to represent the other functions is the most efficient approach. A further advantage of our approach is that no precomputation or no storage of any coefficients is needed. Finally, basing on the results of Section 7.2 we

present our algorithm `computeClassInvariants`(Δ, h, R) in Section 7.3. We show that its bit-complexity is at most $O(h^6 \log h)$.

7.1 Efficient Computation of Fourier Series

In this section we discuss algorithms to compute the Fourier series of the functions j, γ_2, η, and the Weber functions. These series will be used in Section 7.2 to compare running times of different representations to evaluate these functions in practice.

7.1.1 Computing the Fourier Coefficients of the j-function

In this section we discuss an efficient algorithm to determine the Fourier series of the modular function j. It is well known that the j-function is holomorphic on the upper complex half plane $\mathfrak{h}$. Furthermore, j is invariant under the action of $\mathrm{SL}(2,\mathbb{Z})$ and hence periodic with period 1. Thus there is a unique Fourier series with $j(\tau) = \sum_{n=-\infty}^{\infty} c_n \cdot q^n$. Furthermore, as j has a single pole at infinity we have $c_n = 0$ for $n \leq -2$ ([Cox89], Theorem 11.8, p. 225).

To compute the coefficients c_n we make use of recursive formulae due to K. Mahler. Setting 0 for an empty sum we have for all $n \in \mathbb{N}$ ([Mah76], Equation 46, p. 91):

$$c_{4n} = c_{2n+1} + \frac{c_n^2 - c_n}{2} + \sum_{k=1}^{n-1} c_k c_{2n-k} \,, \tag{7.1}$$

$$\begin{aligned} c_{4n+1} &= c_{2n+3} - c_2 c_{2n} + \frac{c_{n+1}^2 - c_{n+1}}{2} + \frac{c_{2n}^2 + c_{2n}}{2} \\ &\quad + \sum_{k=1}^{n} c_k c_{2n-k+2} - \sum_{k=1}^{2n-1} (-1)^{k-1} c_k c_{4n-k} + \sum_{k=1}^{n-1} c_k c_{4n-4k} \,, \end{aligned} \tag{7.2}$$

$$c_{4n+2} = c_{2n+2} + \sum_{k=1}^{n} c_k c_{2n-k+1} \,, \tag{7.3}$$

$$\begin{aligned} c_{4n+3} &= c_{2n+4} - c_2 c_{2n+1} - \frac{c_{2n+1}^2 - c_{2n+1}}{2} \\ &\quad + \sum_{k=1}^{n+1} c_k c_{2n-k+3} - \sum_{k=1}^{2n} (-1)^{k-1} c_k c_{4n-k+2} + \sum_{k=1}^{n} c_k c_{4n-4k+2} \,. \end{aligned} \tag{7.4}$$

However, to make use of the Formulae (7.1) - (7.4), we have to know the coefficients c_{-1}, c_0, c_1, c_2, c_3, and c_5 : Obviously, for $n = 1$ the equations depend on $c_1, \ldots, c_3$. However, (7.2) shows $c_5 = c_5$ in this case, and we have to know c_5 before using Mahler's equations.

Thus we first determine these coefficients by evaluating the representation of j by the normalized Eisenstein series E_4 and E_6 of weight 4 and 6, respectively. The Eisenstein series are defined using the arithmetic function $\sigma_k(n)$, which depends on two parameters $k \in \mathbb{N}_0$ and

$n \in \mathbb{N}$. We have the following definitions and relations ([Kob93], p. 110-112):

$$\sigma_k(n) = \sum_{d|n,d>0} d^k, \qquad k \in \mathbb{N}_0, n \in \mathbb{N}, \tag{7.5}$$

$$E_4(\tau) = 1 + 240 \cdot \sum_{n=1}^{\infty} \sigma_3(n) \cdot q^n, \qquad \tau \in \mathfrak{h}, \tag{7.6}$$

$$E_6(\tau) = 1 - 504 \cdot \sum_{n=1}^{\infty} \sigma_5(n) \cdot q^n, \qquad \tau \in \mathfrak{h}, \tag{7.7}$$

$$j(\tau) = 1728 \cdot \frac{E_4^3(\tau)}{E_4^3(\tau) - E_6^2(\tau)}, \qquad \tau \in \mathfrak{h}. \tag{7.8}$$

We remark that the Fourier series of E_4 and E_6 converge locally uniformly on $\mathfrak{h}$; thus both functions are holomorphic on $\mathfrak{h}$. Hence when performing arithmetic with the Fourier series we do not have to care about convergence or the order of summation.

Let $N \in \mathbb{N}$ be given. To compute $c_{-1}, c_0, \ldots, c_N$ we first compute the Fourier series of the numerator and denominator in (7.8). More precisely, we define sequences $(r_n)_{n\in\mathbb{N}_0}$ and $(\tau_n)_{n\in\mathbb{N}}$ as Fourier coefficients of $E_4^3(\tau)$ and $(E_4^3(\tau) - E_6^2(\tau))/1728$, respectively. We thank Prof. Köhler for pointing out to us this representation of the denominator. We determine r_n for $0 \le n \le N+1$. This can easily be done using Definitions (7.5) and (7.6). Furthermore, we compute the coefficients τ_n for $1 \le n \le N+2$ using a formula of D. Niebur ([Nie75], [Gou97]):

$$\tau_n = n^4 \cdot \sigma_1(n) - 24 \cdot \sum_{k=1}^{n-1} (35k^4 - 52k^3 n + 18k^2 n^2) \cdot \sigma_1(k) \cdot \sigma_1(n-k)\ . \tag{7.9}$$

It is well known that $(E_4^3(\tau) - E_6^2(\tau))/1728$ is up to a constant factor the classical discriminant function Δ on $\mathfrak{h}$ ([Kob93]): $\Delta(\tau) = \frac{(2\pi)^{12}}{1728} \cdot (E_4^3(\tau) - E_6^2(\tau))$. As the discriminant function does not vanish on $\mathfrak{h}$ ([Cox89], Proposition 10.7, p. 206), (7.8) can be transformed to

$$\left(\sum_{n=-1}^{\infty} c_n \cdot q^n \right) \cdot \left(\sum_{n=1}^{\infty} \tau_n \cdot q^n \right) = \sum_{n=0}^{\infty} r_n \cdot q^n\ . \tag{7.10}$$

Thus we get a recursive formula to compute the coefficients c_{n-1} for $0 \le n \le N+1$:

$$\sum_{k=0}^{n} c_{k-1} \cdot \tau_{n+1-k} = r_n. \tag{7.11}$$

Our algorithm `computeCoefficientsOfJViaEisenstein`(N) gets a natural number N as input. It returns the Fourier coefficients c_{-1}, c_0, ..., c_N of the modular function j. The algorithm implements the relation of Equation 7.11. We remark that we make use of $\tau_1 = 1$. Using `computeCoefficientsOfJViaEisenstein`(N) with $N = 5$ we get $c_{-1} = 1$, $c_0 = 744$, $c_1 = 196884$, $c_2 = 21493760$, $c_3 = 864299970$, $c_4 = 20245856256$, and $c_5 = 333202640600$. Finally making use of Mahler's Equations (7.1) - (7.4) we compute the values of c_n up to $n = 50000$ and store them in a file of size 38.7 MByte. The running time on the Pentium III was 9.87 hours. We list some more coefficients: $c_6 = 4252023300096$, $c_7 = 44656994071935$,

Algorithm 7.1: `computeCoefficientsOfJViaEisenstein`(N)

Input: A natural number N.
Output: The Fourier coefficients $c_{-1}, \ldots, c_N$ of the modular function j.

```
Compute r_0, ..., r_{N+1} using the Formulae (7.5) - (7.6);
Compute τ_1, ..., τ_{N+2} using Equation (7.9);
c_{-1} ← 1;
for n = 1 to N + 1 do
    c_{n-1} ← r_n − Σ_{k=0}^{n-1} c_{k-1} · τ_{n+1-k};
return (c_{-1}, ..., c_N);
```

$c_8 = 401490886656000$, $c_9 = 3176440229784420$, and $c_{10} = 22567393309593600$. In addition, in Section A.3 of the appendix we list the coefficients c_n for $49996 \leq n \leq 50000$.

We remark that there are further efficient methods to compute the Fourier coefficients of j. The first one is due to M. Kaneko ([Kan]), who extends work of D. Zagier. We refer to his paper for details. The second one is due to G. Köhler ([Köh02]). Köhler first computes the Fourier coefficients of the function γ_2 as described in Section 7.1.2. He then makes use of the relation $j = \gamma_2^3$.

7.1.2 Computing the Fourier Coefficients of γ_2

In this section we describe how to compute the Fourier coefficients of the function γ_2. We first derive a formula for the Fourier series. The cited arguments are well known. Let $\mathbb{E}$ denote the open unit disc in $\mathbb{C}$, that is $\mathbb{E} = \{z \in \mathbb{C} : |z| < 1\}$. Making use of the Fourier series of j, we define a function $h : \mathbb{E} \to \mathbb{C}$ as follows: $h(q) := \sum_{n=0}^{\infty} c_{n-1} q^n$. Hence we have $j(\tau) = q^{-1} \cdot h(q)$. As j, as a function of q, is meromorphic on $\mathbb{E}$ with a single pole at 0, h is holomorphic on $\mathbb{E}$. Obviously we have $h(0) = 1$ and $h(x) \in \mathbb{R}$ for all $x \in \mathbb{E} \cap \mathbb{R}$. Hence in a neighbourhood U of $q = 0$ there is a holomorphic function $r : U \to \mathbb{C}$ with $r^3(q) = h(q)$ for all $q \in U$, $r(0) = 1$, and $r(x) \in \mathbb{R}$ for all $x \in U \cap \mathbb{R}$. We write $\sum_{n=0}^{\infty} g_n q^n$ for its Laurent expansion at 0. Thus we have $j(\tau) = \left(q^{-\frac{1}{3}} \cdot r(q)\right)^3$ for all $\tau \in \mathfrak{h}$ with $q \in U$. Hence for these values τ the function $q^{-\frac{1}{3}} \cdot r(q)$ is a cube root of j that is real valued for all $\tau \in i\mathbb{R}$ with $q \in U$. On the other hand, Equation (2.17) shows $\mathfrak{f}_2(\tau) \in \mathbb{R}$ for all $\tau \in i\mathbb{R} \cap \mathfrak{h}$. Thus Equation (2.20) yields $\gamma_2(\tau) \in \mathbb{R}$ for these τ. However, γ_2 is also a cube root of j, and $\gamma_2(\tau) = q^{-\frac{1}{3}} \cdot r(q)$ follows for all τ with $q \in U$. As γ_2 is holomorphic on $\mathfrak{h}$ it follows that the Laurent expansion of $r(q)$ is valid on $\mathbb{E}$. Thus for $\tau \in \mathfrak{h}$ we have derived the formula

$$\gamma_2(\tau) = q^{-\frac{1}{3}} \cdot \sum_{n=0}^{\infty} g_n q^n \ . \tag{7.12}$$

Next we present two algorithms for computing the g_n. The first one is similar to the case of the modular function j, that is we present recursive formulae due to Mahler to compute the Fourier coefficients g_n. Second we use an approach due to Köhler ([Köh02]) who derives a recursive formula for the coefficients g_n using the representation $\gamma_2(\tau) = E_4(\tau)/\eta^8(\tau)$. It turns out that the second approach is much faster in practice.

Let us turn to our first approach. Mahler takes a slightly different representation of the Fourier series: He multiplies the term $q^{-\frac{1}{3}}$ into the sum and writes

$$\gamma_2(\tau) = q^{-\frac{1}{3}} + \sum_{n=0}^{\infty} b_{3n+2} q^{\frac{3n+2}{3}}. \tag{7.13}$$

Hence we have $g_0 = 1$ and $g_n = b_{3n-1}$ for $n \geq 1$. If we set 0 for an empty sum we have for all $n \in \mathbb{N}_0$ ([Mah76], Equation 79, p. 115):

$$b_{12n+2} = b_{6n+2} + \sum_{k=0}^{n-1} b_{3k+2} b_{6n-3k-1}\ , \tag{7.14}$$

$$\begin{aligned} b_{12n+5} &= b_{6n+5} - b_2 b_{6n+2} + \frac{b_{3n+2}^2 - b_{3n+2}}{2} + \frac{b_{6n+2}^2 + b_{6n+2}}{2} \\ &\quad + \sum_{k=0}^{n-1} b_{3k+2} b_{6n-3k+2} - \sum_{k=0}^{2n-1} (-1)^{k-1} b_{3k+2} b_{12n-3k+2} + \sum_{k=0}^{n-1} b_{3k+2} b_{12n-12k-4}\ , \end{aligned} \tag{7.15}$$

$$b_{12n+8} = b_{6n+5} + \sum_{k=0}^{n-1} b_{3k+2} b_{6n-3k+2} + \frac{b_{3n+2}^2 - b_{3n+2}}{2}\ , \tag{7.16}$$

$$\begin{aligned} b_{12n+11} &= b_{6n+8} - b_2 b_{6n+5} - \frac{b_{6n+5}^2 - b_{6n+5}}{2} \\ &\quad + \sum_{k=0}^{n} b_{3k+2} b_{6n-3k+5} - \sum_{k=0}^{2n} (-1)^{k-1} b_{3k+2} b_{12n-3k+8} + \sum_{k=0}^{n} b_{3k+2} b_{12n-12k+2}\ . \end{aligned} \tag{7.17}$$

Evaluating these equations for $n = 0$ we have $b_2 = b_2$, $b_5 = b_5$, $b_8 = b_5 + (b_2^2 - b_2)/2$ and $b_{11} = b_8 - b_2 b_5 - (b_5^2 - b_5)/2 + b_2 b_5 + b_2 b_8 + b_2^2$. Hence b_2 and b_5 uniquely determine all coefficients b_n. Equation (7.12) yields

$$j(\tau) = \gamma_2^3(\tau) = q^{-1} \cdot (1 + 3g_1 q + 3(g_1^2 + g_2) q^2 + O(q^3)). \tag{7.18}$$

Thus comparing (7.18) with the Fourier series of the $j-$function we get $g_1 = c_0/3$ and $g_2 = c_1/3 - g_1^2$, hence $b_2 = g_1 = 248$ and $b_5 = g_2 = 4124$. Using $g_n = b_{3n-1}$ for $n \geq 3$, we computed the coefficients g_n up to $n = 50000$. The memory to store these coefficients is 22.3 MByte. The computation took us 5.23 hours on the Pentium III.

Now we turn to Köhler's approach. He makes use of the representation $\gamma_2(\tau) = E_4(\tau)/\eta^8(\tau)$. This representation is an easy consequence of Formula (7.8) and the well known relations

$$\Delta(\tau) = \frac{(2\pi)^{12}}{1728} \cdot (E_4^3(\tau) - E_6^2(\tau)) = (2\pi)^{12} \cdot \eta^{24}(\tau)$$

(see for instance [Apo90], Theorem 3.3, p. 51). Köhler observes that $\eta^8(\tau)$ may be written as a Hecke theta series (see [Sch53], [Ser85], and [Köh88]). More precisely, the function $\eta^8(\tau)$ may be written as a series

$$\eta^8(\tau) = q^{1/3} \cdot \sum_{n=0}^{\infty} c_8(n) q^n\ , \tag{7.19}$$

where the coefficients $c_8(n)$ are given by

$$c_8(n) = \sum_{\substack{x\in\mathbb{N},y\in\mathbb{N}_0\\ x^2+xy+y^2=3n+1}} \left(\frac{x-y}{3}\right)\cdot(x+y\cdot e^{2\pi i/6})^3 \ . \tag{7.20}$$

As usual $\left(\frac{\cdot}{\cdot}\right)$ denotes the Jacobi symbol. Our task is to determine the coefficients g_n, $0 \le n \le N$, for some $N \in \mathbb{N}$. We assume that the coefficients $c_8(0),\ldots,c_8(N)$ are already known (algorithm `computeCoefficientsOfEtaTo8`(N) described below computes these coefficients). The representation $\gamma_2(\tau) = E_4(\tau)/\eta^8(\tau)$ and the Fourier series in (7.12), (7.6), and (7.19) yield

$$\sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} g_k c_8(n-k)\right)q^n \quad = \quad 1+240\cdot\sum_{n=1}^{\infty}\sigma_3(n)\cdot q^n \ . \tag{7.21}$$

Making use of $c_8(0)=1$ we conclude $g_0=1$ and $g_n = 240\sigma_3(n) - \sum_{k=0}^{n-1} g_k c_8(n-k)$ for $n \ge 1$.

It remains to explain algorithm `computeCoefficientsOfEtaTo8`(N), which again is due to G. Köhler ([Köh02]). Input of the algorithm is some $N \in \mathbb{N}$, and it returns the coefficients $c_8(n)$ for $0 \le n \le N$. In order to get the coefficient $c_8(n)$ we have to find all pairs $(x,y) \in \mathbb{N}\times\mathbb{N}_0$ with $x \not\equiv y \bmod 3$ and $x^2+xy+y^2 = 3n+1$. However, in order to avoid solving this equation, Köhler takes a different, brute-force like approach. The first observation is that for all $(x,y) \in \mathbb{N}\times\mathbb{N}_0$ with $x \not\equiv y \bmod 3$ we have $x^2+xy+y^2 \equiv 1 \bmod 3$. Hence each such pair yields a contribution to $c_8((x^2+xy+y^2-1)/3)$. Second, if $(x,y) \in \mathbb{N}\times\mathbb{N}$, it is easy to that

$$\begin{aligned}\left(\frac{x-y}{3}\right)\cdot(x+y\cdot e^{2\pi i/6})^3 \quad &+ \quad \left(\frac{y-x}{3}\right)\cdot(y+x\cdot e^{2\pi i/6})^3\\ &= \quad \left(\frac{x-y}{3}\right)\cdot(2x^3+3x^2y-3xy^2-2y^3)\\ &= \quad \left(\frac{x-y}{3}\right)\cdot(x-y)\cdot(2(x^2+xy+y^2)+3xy) \ .\end{aligned}$$

Hence, if $y \neq 0$, it is sufficient to take pairs (x,y) with $x > y$ into account.

Algorithm `computeCoefficientsOfEtaTo8`(N) proceeds as follows: For all $0 \le n \le N$ it first initializes $c_8(n)$ with 0. Then, it goes through all relevant pairs $(x,0)$ with $1 \le x \le \sqrt{3N+1}$. We distinguish the cases $x \equiv 1 \bmod 3$ and $x \equiv 2 \bmod 3$ with corresponding Jacobi symbols $\left(\frac{x}{3}\right)$ equal to 1 and -1, respectively. The respective contribution $\pm x^3$ is added to the coefficient $c_8((x^2-1)/3)$. Next the algorithm considers all pairs $(x,y) \in \mathbb{N}\times\mathbb{N}$ with $x > y$ and $x \not\equiv y \bmod 3$. We first fix y with $1 \le y \le \sqrt{3N+1}$. Then we pass through all x with $y < x \le -y/2 + \sqrt{3N+1-3y^2/4}$ and add the corresponding contribution to $c_8((x^2+xy+y^2-1)/3)$. Obviously, `computeCoefficientsOfEtaTo8`(N) is correct.

It turns out that this approach is much more efficient in practice than using the Mahler formulae. For instance, the computation of the coefficients g_n up to $n = 50000$ only takes about an hour on the Pentium III. Some coefficients g_n are listed in Table 7.1.

7.1.3 Computing the Fourier Coefficients of the $\eta-$function

In this section we present our algorithm `computeCoefficientsOfEta`. This algorithm computes the Fourier coefficients of the $\eta-$function. However, as before, we first derive a formula for the Fourier coefficients.

Algorithm 7.2: `computeCoefficientsOfEtaTo8`(N)

Input: A natural number N.
Output: The Fourier coefficients $c_8(0), \ldots, c_8(N)$ of $\eta^8(\tau)$.

for $n = 0$ to N **do**
  $c_8(n) \leftarrow 0$; //initialize all coefficients $c_8(n)$ with 0
$x \leftarrow 1$; //go through all relevant pairs $(x, 0)$ with $x \equiv 1 \bmod 3$ and thus $\left(\frac{x}{3}\right) = 1$
while $x \leq \lfloor\sqrt{3N+1}\rfloor$ **do**
  $m \leftarrow x^2$; $c_8((m-1)/3) \leftarrow c_8((m-1)/3) + x \cdot m$; $x \leftarrow x + 3$;
$x \leftarrow 2$; //go through all relevant pairs $(x, 0)$ with $x \equiv 2 \bmod 3$ and hence $\left(\frac{x}{3}\right) = -1$
while $x \leq \lfloor\sqrt{3N+1}\rfloor$ **do**
  $m \leftarrow x^2$; $c_8((m-1)/3) \leftarrow c_8((m-1)/3) - x \cdot m$; $x \leftarrow x + 3$;
for $y = 1$ to $\lfloor\sqrt{3N+1}\rfloor$ **do**
  for $x = y + 1$ to $\lfloor -y/2 + \sqrt{3N + 1 - 3y^2/4}\rfloor$ **do**
    if $x \not\equiv y \bmod 3$ **then**
      $m \leftarrow x^2 + xy + y^2$;
      $c_8((m-1)/3) \leftarrow c_8((m-1)/3) + \left(\frac{x-y}{3}\right) \cdot (x - y) \cdot (2m + 3xy)$;
return $(c_8(0), \ldots, c_8(N))$;

g_0	1	g_{11}	1749556736	g_{22}	24551042107480
g_1	248	g_{12}	4848776870	g_{23}	51301080086528
g_2	4124	g_{13}	12908659008	g_{24}	105561758786885
g_3	34752	g_{14}	33161242504	g_{25}	214100032685072
g_4	213126	g_{15}	82505707520	g_{26}	428374478862400
g_5	1057504	g_{16}	199429765972	g_{27}	846173187465216
g_6	4530744	g_{17}	469556091240	g_{28}	1651298967150546
g_7	17333248	g_{18}	1079330385764	g_{29}	3185652564830016
g_8	60655377	g_{19}	2426800117504	g_{30}	6078963644150128
g_9	197230000	g_{20}	5346409013164	g_{31}	11480231806541824
g_{10}	603096260	g_{21}	11558035326944	g_{32}	21467177880529689

Table 7.1: Some Fourier coefficients of the function γ_2

The main result we make use of is due to Euler. It states

$$\eta(\tau) \quad = \quad q^{\frac{1}{24}} \sum_{n\in\mathbb{Z}} (-1)^n q^{\frac{3n^2+n}{2}}. \tag{7.22}$$

Multiplying $q^{\frac{1}{24}}$ into the sum and using some properties of the Jacobi symbol $\left(\frac{\cdot}{\cdot}\right)$, we get

$$\begin{aligned} \eta(\tau) &= \sum_{n\in\mathbb{Z}} (-1)^n q^{\frac{(6n+1)^2}{24}} && (7.23)\\ &= \sum_{n=1}^{\infty} \left(\frac{12}{n}\right) q^{\frac{n^2}{24}} && (7.24) \end{aligned}$$

(we thank Prof. Köhler for pointing out to us this relation). Equation (7.22) shows that we can define a sequence $(e_n)_{n\in\mathbb{N}_0}$ by $\eta(\tau) = q^{\frac{1}{24}} \sum_{n=0}^{\infty} e_n q^n$. We prove the following result:

Proposition 7.1.1 *Let e_n be as defined above. Then we have for all $n \in \mathbb{N}_0$*

$$e_n = \begin{cases} 0, & \text{if} \quad 24n+1 \text{ is not a square in } \mathbb{Z}, \\ \left(\frac{12}{\sqrt{24n+1}}\right), & \text{if} \quad 24n+1 \text{ is a square in } \mathbb{Z}. \end{cases}$$

Proof: Equation (7.24) and the definition of the e_n yield $\sum_{m=1}^{\infty} \left(\frac{12}{m}\right) q^{\frac{m^2}{24}} = \sum_{n=0}^{\infty} e_n q^{\frac{24n+1}{24}}$. Comparing the coefficients, we get $e_n = \left(\frac{12}{\sqrt{24n+1}}\right)$, if $m^2 = 24n+1$, and $e_n = 0$ otherwise. □

Proposition 7.1.1 shows that we have $e_n \in \{-1, 0, 1\}$. Furthermore, as the exponents in (7.22) grow quadratically coefficients $e_n \neq 0$ are rather sparse. In order to compute the coefficients e_n, we make use of Equation (7.23). Thus we can avoid the computation of square roots and Jacobi symbols.

We turn to our algorithm `computeCoefficientsOfEta`(N). Again we thank Prof. Köhler for helpful comments. The algorithm gets $N \in \mathbb{N}$ as input and returns the coefficients $e_0, \ldots, e_N$. The idea is simply to initialize all e_n with 0 and assign ± 1 to the appropriate coefficients. The algorithm first computes the coefficients $e_n \neq 0$ for positive n in Equation (7.24). In our algorithm, we set $i(k) = (6k+1)^2$. Thus we have $e_{(i-1)/24} = (-1)^k$. In addition, we remark that $i(k+1) - i(k) = 72k + 48$. Let $b = \lfloor(\sqrt{24N+1} - 1)/6\rfloor$. We have to compute the coefficients $e_{(i-1)/24}$ for all k with $0 \leq k \leq b$. As we compute two coefficients e_n in one cycle, we have to distinguish the cases of even and odd b, respectively. Then, the algorithm computes the coefficients e_n for negative n in Equation (7.23). In this case we set $i(k) = (6k-1)^2$. Thus we have $i(k+1) - i(k) = 72k + 24$. Again, we have to distinguish the cases of even and odd b, respectively.

A sample running of `computeCoefficientsOfEta`(N) with $N = 5.000.000$ took us 0.36 seconds on the Pentium III. The memory to store these coefficients is 9.54 MByte. The first coefficients e_n can be found in Table 7.2.

Algorithm 7.3: `computeCoefficientsOfEta`(N)

Input: A natural number N.
Output: The coefficients $e_0, \ldots, e_N$ of the η−function.

for $n = 0$ to N **do**
  $e_n \leftarrow 0$;
$b \leftarrow \lfloor(\sqrt{24N+1} - 1)/6\rfloor$; $b_e \leftarrow$ `false`;
if $2 \mid b$ **then**
  $b \leftarrow b - 1$; $b_e \leftarrow$ `true`;
$k \leftarrow 0$; $i \leftarrow 1$; //we have $i = (6k+1)^2$
while $k < b$ **do**
  $e_{(i-1)/24} \leftarrow 1$; $i \leftarrow i + 72k + 48$; $k \leftarrow k + 1$;
  $e_{(i-1)/24} \leftarrow -1$; $i \leftarrow i + 72k + 48$; $k \leftarrow k + 1$;
if $b_e =$ `true` **then**
  $e_{(i-1)/24} \leftarrow 1$;
$b \leftarrow \lfloor(\sqrt{24N+1} + 1)/6\rfloor$; $b_e \leftarrow$ `true`;
if $2 \nmid b$ **then**
  $b \leftarrow b - 1$; $b_e \leftarrow$ `false`;
$k \leftarrow 1$; $i \leftarrow 25$; //we have $i = (6k-1)^2$
while $k < b$ **do**
  $e_{(i-1)/24} \leftarrow -1$; $i \leftarrow i + 72k + 24$; $k \leftarrow k + 1$;
  $e_{(i-1)/24} \leftarrow 1$; $i \leftarrow i + 72k + 24$; $k \leftarrow k + 1$;
if $b_e =$ `false` **then**
  $e_{(i-1)/24} \leftarrow -1$;
return $(e_0, \ldots, e_N)$;

e_0	1	e_{11}	0	e_{22}	1	e_{33}	0	e_{44}	0	e_{55}	0
e_1	-1	e_{12}	-1	e_{23}	0	e_{34}	0	e_{45}	0	e_{56}	0
e_2	-1	e_{13}	0	e_{24}	0	e_{35}	-1	e_{46}	0	e_{57}	1
e_3	0	e_{14}	0	e_{25}	0	e_{36}	0	e_{47}	0	e_{58}	0
e_4	0	e_{15}	-1	e_{26}	1	e_{37}	0	e_{48}	0	e_{59}	0
e_5	1	e_{16}	0	e_{27}	0	e_{38}	0	e_{49}	0	e_{60}	0
e_6	0	e_{17}	0	e_{28}	0	e_{39}	0	e_{50}	0	e_{61}	0
e_7	1	e_{18}	0	e_{29}	0	e_{40}	-1	e_{51}	1	e_{62}	0
e_8	0	e_{19}	0	e_{30}	0	e_{41}	0	e_{52}	0	e_{63}	0
e_9	0	e_{20}	0	e_{31}	0	e_{42}	0	e_{53}	0	e_{64}	0
e_{10}	0	e_{21}	0	e_{32}	0	e_{43}	0	e_{54}	0	e_{65}	0

Table 7.2: Some Fourier coefficients e_n of the η−function

7.1.4 Computing the Fourier Coefficients of the Weber function $\mathfrak{f}_2$

In this section we show how to efficiently compute the Fourier coefficients of the Weber function $\mathfrak{f}_2$. Then, in Section 7.1.5 we use well known formulae connecting the functions $\mathfrak{f}_2$, $\mathfrak{f}_1$ and $\mathfrak{f}$ to compute the Fourier coefficients of $\mathfrak{f}_1$ and $\mathfrak{f}$. However, let us first turn to the function $\mathfrak{f}_2$.

The main task in computing the coefficients of the Fourier series of $\mathfrak{f}_2$ is to determine the representation of the infinite product in (2.17) by a power series in q. Obviously the product in (2.17) can uniquely be written as a Fourier series $\sum_{n=0}^{\infty} f_{2,n}q^n$. An easy calculation shows that for $n \geq 0$ we have

$$f_{2,n} = \#\{(a_1, ..., a_l) \in \mathbb{N}_0^l : l > 0, a_1 < a_2 < ... < a_l, \sum_{j=1}^{l} a_j = n\}. \tag{7.25}$$

However, our tests indicate that (7.25) is infeasible to compute $f_{2,n}$ for $n > 140$ in practice.

Instead, we make use of the Fourier series of the $\eta-$function from Section 7.1.3. Using (2.14) and (2.17) one easily sees $\mathfrak{f}_2(\tau) = \sqrt{2} \cdot \frac{\eta(2\tau)}{\eta(\tau)}$. In Section 7.1.3 we derived the formula $\eta(\tau) = q^{\frac{1}{24}} \sum_{n=0}^{\infty} e_n q^n$. As the η-function does not vanish on $\mathfrak{h}$ we use the Fourier series of $\mathfrak{f}_2$ and η to get

$$\begin{aligned} \sum_{n=0}^{\infty} e_n q^{2n} &= \sum_{n=0}^{\infty} f_{2,n}q^n \cdot \sum_{n=0}^{\infty} e_n q^n \\ &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} f_{2,k} e_{n-k} \right) \cdot q^n. \end{aligned} \tag{7.26}$$

Hence, making use of $e_0 = 1$, Equation (7.26) yields for $n \in \mathbb{N}_0$

$$f_{2,n} = \begin{cases} -\sum_{k=0}^{n-1} f_{2,k} e_{n-k}, & if \quad 2 \nmid n, \\ e_{\frac{n}{2}} - \sum_{k=0}^{n-1} f_{2,k} e_{n-k}, & if \quad 2 \mid n. \end{cases}$$

Our algorithm `computeCoefficientsOfF2`$(N, (e_0, ..., e_N))$ gets a natural number N and the first $N+1$ Fourier coefficients of the η-function as input; the algorithm returns the coefficients $f_{2,0}, \ldots, f_{2,N}$.

According to our experience this is the most efficient way in practice to compute the Fourier coefficients of $\mathfrak{f}_2$. As in the case of the j- and the γ_2-function we determined all coefficients $f_{2,n}$ up to $n = 50000$ using `computeCoefficientsOfF2`, which took us 22 minutes on the Pentium III. The necessary amount of storage is 5.48 MByte. In Table 7.3 we list the first coefficients $f_{2,n}$.

We remark an alternative approach to determine the coefficients $f_{2,n}$. Fix a natural number N. Obviously $f_{2,N}$ is the coefficient of X^N of the polynomial $\prod_{k=1}^{N}(1 + X^k)$. However, for example to get the coefficient $f_{2,1700}$ the polynomial is of degree 1445850. The CPU time to get the coefficients $f_{2,n}$ up to $N = 1700$ was 305.7 hours on the SUN UltraSPARC-IIi.

Algorithm 7.4: `computeCoefficientsOfF2`$(N, (e_0, ..., e_N))$

Input: An upper bound $N \in \mathbb{N}$.
The Fourier coefficients $e_0, \ldots, e_N$ of the $\eta-$function.
Output: The coefficients $f_{2,n}$ for $0 \leq n \leq N$.

$f_{2,0} \leftarrow 1$;
for $n = 1$ to N **do**
  if $2 \mid n$ **then**
    $f_{2,n} \leftarrow e_{\frac{n}{2}} - \sum_{k=0}^{n-1} f_{2,k} e_{n-k}$;
  else
    $f_{2,n} \leftarrow -\sum_{k=0}^{n-1} f_{2,k} e_{n-k}$;
return $(f_{2,0}, \ldots, f_{2,N})$;

$f_{2,0}$	1	$f_{2,11}$	12	$f_{2,22}$	89	$f_{2,33}$	448	$f_{2,44}$	1816	$f_{2,55}$	6378
$f_{2,1}$	1	$f_{2,12}$	15	$f_{2,23}$	104	$f_{2,34}$	512	$f_{2,45}$	2048	$f_{2,56}$	7108
$f_{2,2}$	1	$f_{2,13}$	18	$f_{2,24}$	122	$f_{2,35}$	585	$f_{2,46}$	2304	$f_{2,57}$	7917
$f_{2,3}$	2	$f_{2,14}$	22	$f_{2,25}$	142	$f_{2,36}$	668	$f_{2,47}$	2590	$f_{2,58}$	8808
$f_{2,4}$	2	$f_{2,15}$	27	$f_{2,26}$	165	$f_{2,37}$	760	$f_{2,48}$	2910	$f_{2,59}$	9792
$f_{2,5}$	3	$f_{2,16}$	32	$f_{2,27}$	192	$f_{2,38}$	864	$f_{2,49}$	3264	$f_{2,60}$	10880
$f_{2,6}$	4	$f_{2,17}$	38	$f_{2,28}$	222	$f_{2,39}$	982	$f_{2,50}$	3658	$f_{2,61}$	12076
$f_{2,7}$	5	$f_{2,18}$	46	$f_{2,29}$	256	$f_{2,40}$	1113	$f_{2,51}$	4097	$f_{2,62}$	13394
$f_{2,8}$	6	$f_{2,19}$	54	$f_{2,30}$	296	$f_{2,41}$	1260	$f_{2,52}$	4582	$f_{2,63}$	14848
$f_{2,9}$	8	$f_{2,20}$	64	$f_{2,31}$	340	$f_{2,42}$	1426	$f_{2,53}$	5120	$f_{2,64}$	16444
$f_{2,10}$	10	$f_{2,21}$	76	$f_{2,32}$	390	$f_{2,43}$	1610	$f_{2,54}$	5718	$f_{2,65}$	18200

Table 7.3: Some Fourier coefficients $f_{2,n}$ of the Weber function $\mathfrak{f}_2$

7.1.5 Computing the Fourier Series of the Weber functions $\mathfrak{f}_1$ and $\mathfrak{f}$

We describe algorithms to compute the Fourier coefficients of the Weber functions $\mathfrak{f}_1$ and $\mathfrak{f}$, respectively. It turns out that up to sign the coefficients of both Fourier series are equal. Hence, we will only present our algorithm `computeCoefficientsOfF1` to determine the Fourier coefficients of $\mathfrak{f}_1$. In order to describe algorithm `computeCoefficientsOfF1`, we first derive a formula which links the Fourier coefficients of $\mathfrak{f}_2$ and $\mathfrak{f}_1$.

Thus let us turn to the Weber function $\mathfrak{f}_1$. As in the case of $\mathfrak{f}_2$, we have to represent the infinite product in (2.16) by a power series in q. Again the infinite product is holomorphic and periodic on $\mathfrak{h}$; however, as a function of τ its period is 2. Thus the product can be written uniquely as a Fourier series $\sum_{n=-\infty}^{\infty} f_{1,n} q^{n/2}$. Furthermore, Equation (2.16) shows $f_{1,n} = 0$ for all $n < 0$. We next derive a formula, which will link the Fourier series of $\mathfrak{f}$ and $\mathfrak{f}_1$.

Proposition 7.1.2 *For $n \geq 0$ the coefficients $f_{1,n}$ satisfy*

$$f_{1,n} = (-1)^n \cdot \#\{(a_1, ..., a_l) \in \mathbb{N}^l : l > 0, a_1 < a_2 < ... < a_l, \sum_{j=1}^{l} (2a_j - 1) = n\}. \qquad (7.27)$$

Proof: We have $\prod_{n=1}^{\infty}(1 - q^{n-\frac{1}{2}}) = \prod_{n=1}^{\infty}(1 - q^{\frac{2n-1}{2}}) = \sum_{n=0}^{\infty} f_{1,n} \cdot q^{\frac{n}{2}}$. Fix $n \in \mathbb{N}_0$. We want to determine the coefficient $f_{1,n}$. By a contribution to $f_{1,n}$ we mean a sequence of powers $q^{(2a_i-1)/2}$, such that all a_i are positive, distinct, and the product of all these powers equals $q^{\frac{n}{2}}$. The length of a contribution is the number of q-powers in the sequence. The product of the coefficients of a contribution equals +1, if and only if its length is even, and -1 otherwise. However, if n is even, the length of a contribution has to be even, too, and odd otherwise. This proves the proposition. □

As in the case of the Weber function $\mathfrak{f}_2$ we were not able to determine $f_{1,n}$ for $n > 140$ in reasonable time using Proposition 7.1.2. Thus we make use of the following formula connecting $\mathfrak{f}_1$ and $\mathfrak{f}_2$ ([Cox89], p. 256):

$$\mathfrak{f}_1(\tau) \cdot \mathfrak{f}_2\left(\frac{\tau}{2}\right) = \sqrt{2}\,. \qquad (7.28)$$

We make use of Equation (2.16), (2.17), and (7.28) to get

$$\left(\sum_{n=0}^{\infty} f_{1,n} q^{\frac{n}{2}}\right) \cdot \left(\sum_{n=0}^{\infty} f_{2,n} q^{\frac{n}{2}}\right) = 1. \qquad (7.29)$$

Both q-expansions converge locally uniformly on $\mathfrak{h}$. Thus their product series converges locally uniformly, too, and the order of summation does not matter. Furthermore, both sides of (7.29) are holomorphic and periodic functions with period 2 on $\mathfrak{h}$. Comparing their Fourier series we get

$$f_{1,0} \cdot f_{2,0} = 1, \qquad (7.30)$$

$$\sum_{j=0}^{n} f_{1,n-j} \cdot f_{2,j} = 0, \qquad 1 \leq n. \qquad (7.31)$$

Thus knowing the coefficients $f_{2,0} = 1, \ldots, f_{2,N}$ we get a recursive formula to compute $f_{1,0}, \ldots, f_{1,N}$:

$$f_{1,0} = 1, \tag{7.32}$$

$$f_{1,n} = -\sum_{j=1}^{n} f_{1,n-j} \cdot f_{2,j}, \qquad 1 \leq n. \tag{7.33}$$

Our algorithm `computeCoefficientsOfF1`$(N, (f_{2,0}, ..., f_{2,N}))$ gets a natural number N and an array $(f_{2,0}, ..., f_{2,N})$ consisting of the Fourier coefficients $f_{2,n}$ as input. The algorithm returns an array $(f_{1,0}, ..., f_{1,N})$ of length $N + 1$ storing the coefficients $f_{1,n}$.

Algorithm 7.5: `computeCoefficientsOfF1`$(N, (f_{2,0}, ..., f_{2,N}))$

Input: An upper bound $N \in \mathbb{N}$.
An array $(f_{2,0}, ..., f_{2,N})$ storing the Fourier coefficients of $\mathfrak{f}_2$.
Output: An array $(f_{1,0}, ..., f_{1,N})$ storing the Fourier coefficients of $\mathfrak{f}_1$.

$f_{1,0} \leftarrow 1$;
for $n = 1$ to N **do**
  $f_{1,n} \leftarrow -\sum_{j=1}^{n} f_{1,n-j} \cdot f_{2,j}$;
return $(f_{1,0}, ..., f_{1,N})$;

For $N = 50000$ our algorithm `computeCoefficientsOfF1` terminates after about 76 minutes on the Pentium III. The amount of storage of the coefficients $f_{1,0}, ..., f_{1,50000}$ is 3.87 MByte. Table 7.4 lists some coefficients $f_{1,n}$.

$f_{1,0}$	1	$f_{1,11}$	-2	$f_{1,22}$	8	$f_{1,33}$	-25	$f_{1,44}$	63	$f_{1,55}$	-144
$f_{1,1}$	-1	$f_{1,12}$	3	$f_{1,23}$	-9	$f_{1,34}$	26	$f_{1,45}$	-68	$f_{1,56}$	157
$f_{1,2}$	0	$f_{1,13}$	-3	$f_{1,24}$	11	$f_{1,35}$	-29	$f_{1,46}$	72	$f_{1,57}$	-168
$f_{1,3}$	-1	$f_{1,14}$	3	$f_{1,25}$	-12	$f_{1,36}$	33	$f_{1,47}$	-78	$f_{1,58}$	178
$f_{1,4}$	1	$f_{1,15}$	-4	$f_{1,26}$	12	$f_{1,37}$	-35	$f_{1,48}$	87	$f_{1,59}$	-192
$f_{1,5}$	-1	$f_{1,16}$	5	$f_{1,27}$	-14	$f_{1,38}$	37	$f_{1,49}$	-93	$f_{1,60}$	209
$f_{1,6}$	1	$f_{1,17}$	-5	$f_{1,28}$	16	$f_{1,39}$	-41	$f_{1,50}$	98	$f_{1,61}$	-223
$f_{1,7}$	-1	$f_{1,18}$	5	$f_{1,29}$	-17	$f_{1,40}$	46	$f_{1,51}$	-107	$f_{1,62}$	236
$f_{1,8}$	2	$f_{1,19}$	-6	$f_{1,30}$	18	$f_{1,41}$	-49	$f_{1,52}$	117	$f_{1,63}$	-255
$f_{1,9}$	-2	$f_{1,20}$	7	$f_{1,31}$	-20	$f_{1,42}$	52	$f_{1,53}$	-125	$f_{1,64}$	276
$f_{1,10}$	2	$f_{1,21}$	-8	$f_{1,32}$	23	$f_{1,43}$	-57	$f_{1,54}$	133	$f_{1,65}$	-294

Table 7.4: Some Fourier coefficients $f_{1,n}$ of the weber function $\mathfrak{f}_1$

Finally, we turn to the Fourier coefficients of the Weber function $\mathfrak{f}$. Again, as in the case of the Weber functions $\mathfrak{f}_1$ and $\mathfrak{f}_2$, the infinite product in Equation (2.15) can be written uniquely as a Fourier series $\sum_{n=0}^{\infty} f_n q^{\frac{n}{2}}$. We first give a formula for f_n, which is up to sign the same as for $f_{1,n}$.

Proposition 7.1.3 *For $n \geq 0$ the coefficients f_n satisfy*

$$f_n = \#\{(a_1, ..., a_l) \in \mathbb{N}^l : l > 0, a_1 < a_2 < ... < a_l, \sum_{j=1}^{l}(2a_j - 1) = n\}. \tag{7.34}$$

Proof: The proof is the same as in the case of $\mathfrak{f}_1$ except for the fact that any contribution to f_n yields a value $+1$. □

Proposition 7.1.2 and 7.1.3 show the relationship $f_n = (-1)^n \cdot f_{1,n}$ for all $n \in \mathbb{Z}$. Thus for given n knowing f_n is equivalent to knowing $f_{1,n}$.

7.2 Computation of Class Invariants Using Different Representations

This section deals with comparing alternative methods to compute class invariants. More precisely, let an imaginary quadratic discriminant Δ and a reduced representative $Q \in C(\Delta)$ be given. Furthermore, fix a floating point precision F and a number-theoretical function $g \in \{\mathfrak{f}, \mathfrak{f}_1, \mathfrak{f}_2, \gamma_2, j\}$. We show that in practice the computation of $g(\tau_Q)$ within the precision F using an efficient representation of the η-function is by far the most efficient alternative. Furthermore, we present running times for all alternatives we make use of.

In order to avoid mentioning the floating precision F all the time, we simply say that we compute $g(\tau_Q)$. However, as we have to implement our algorithms, we always mean that we compute $g(\tau_Q)$ within the precision F.

The formulae and relations of the functions involved make the following proceeding plausible: As the functions j, γ_2, and the Weber functions may be expressed in terms of Dedekind's η-function, we first compare in Section 7.2.1 three different approaches to determine the value $\eta(\tau_Q)$. We show that the representation of η by its Euler sum (7.22) is the most efficient way in practice to compute $\eta(\tau_Q)$. Next, in Section 7.2.2, we explore alternatives of computing $\mathfrak{f}_2(\tau_Q)$ by its product formula (2.17), its Fourier series from Section 7.1.4, and finally by the term $\mathfrak{f}_2(\tau_Q) = \sqrt{2}\frac{\eta(2\tau_Q)}{\eta(\tau_Q)}$; in the last expression we make use of our results on the η-function. The result of Section 7.2.2 is that the last representation is most efficient one to evaluate $\mathfrak{f}_2(\tau_Q)$ in practice. Similarly, we deal in Section 7.2.3 with $\mathfrak{f}_1$ and in Section 7.2.4 with $\mathfrak{f}$, respectively. Again, a representation of $\mathfrak{f}_1$ and $\mathfrak{f}$ using the η-function turns out to be superior, respectively.

Finally, in Section 7.2.5 we discuss the computation of $\gamma_2(\tau_Q)$ and $j(\tau_Q)$, respectively. The result of Section 7.2.5 is that using the Weber function $\mathfrak{f}_2$ is in practice the most efficient alternative to compute $\gamma_2(\tau_Q)$ using Equation (2.20). Thus as $j = \gamma_2^3$ the same is true for the modular function j.

7.2.1 Efficient Computation of the η-function

In this section we compare different representations of Dedekind's η-function to compute the value $\eta(\tau)$ in practice, if $\tau \in \mathfrak{h}$ is given. From Section 7.1.3 we know three different ways to represent $\eta(\tau)$:

- First, we make use of the product formula $q^{\frac{1}{24}} \cdot \prod_{n=1}^{\infty}(1-q^n)$, that is we use the defining Equation (2.14).
- Second, we compute $\eta(\tau)$ by evaluating its Fourier series $q^{\frac{1}{24}} \sum_{n=0}^{\infty} e_n q^n$ from Section 7.1.3.
- Finally, we use the Euler sum $q^{\frac{1}{24}} \sum_{n\in\mathbb{Z}} (-1)^n q^{\frac{3n^2+n}{2}}$ from Equation (7.22).

The corresponding algorithms are `computeEtaViaProduct`, `computeEtaViaFourierSeries`, and `computeEtaViaEulerSum`, respectively. We show that `computeEtaViaEulerSum` turns out to be the most efficient way in practice to compute $\eta(\tau)$ within a given floating point precision F.

Input of all three algorithms is a complex number τ in the upper complex half plane $\mathfrak{h}$. In addition, the algorithms require a floating point precision F. Each algorithm returns the value $\eta(\tau)$ within the precision F, that is either algorithm returns a complex number η_τ with $|\eta_\tau - \eta(\tau)| < 10^{-F}$.

We first describe our algorithm `computeEtaViaProduct`(τ, F). It is a straightforward consequence of the defining equation (2.14) of the η-function. Given τ and F, the algorithm first computes $q = e^{2\pi i\tau}$ within the precision F. For $n \in \mathbb{N}$ we use the variable q_n to store the power q^n, that is we have $q_n = q^n$. In addition, throughout the algorithm we store the current value of $\prod_{n=1}^{N}(1-q^n)$ in η; thus η stores the current approximation of $\eta(\tau)/q^{\frac{1}{24}}$. As the convergence behavior of this product is not obvious, we introduce an array η_l of length 5 to store the five previous values of η. More precisely, if $N \geq 6$ and $\eta = \prod_{n=1}^{N}(1-q^n)$, we have $\eta_l[k] = \prod_{n=1}^{N-5+k}(1-q^n)$ for $0 \leq k \leq 4$. We are convinced that η is the correct value of $\eta(\tau)/q^{\frac{1}{24}}$ if $|\eta - \eta_l[k]| < 10^{-F}$ for all $0 \leq k \leq 4$. We introduce a boolean η_b to check this condition. The practical performance of `computeEtaViaProduct` for some sample input may be found in Figure 7.1 and Table 7.5, respectively.

We next discuss our second algorithm `computeEtaViaFourierSeries`(τ, F). This algorithm makes use of the Fourier series of the η-function; this series was introduced in Section 7.1.3. To sum up we showed that $\eta(\tau) = q^{\frac{1}{24}} \sum_{n=0}^{\infty} e_n q^n$ with $e_n \in \{-1, 0, 1\}$. In addition, we stated that this series is sparse, that is coefficients with $e_n \neq 0$ are rare. We assume that sufficiently many coefficients are to our disposal; for instance, we make use of $e_0, \ldots, e_{5000000}$.

As above the algorithm first computes $q = e^{2\pi i\tau}$ within the precision F, and again we set $q_n = q^n$ for $n \in \mathbb{N}_0$. Furthermore, we store the current value of $\sum_{n=0}^{N} e_n q^n$ in η. Now let $n \in \mathbb{N}_0$ with $e_n \neq 0$. As the series is rather sparse, we store the value of n in l. In order to avoid subsequent multiplications by q, we increase n by 1 until $e_n \neq 0$. Then we compute the current value q_n by multiplying q_l with q^{n-l}; furthermore, we compute the new value of η. As the Fourier series is sparse we are convinced to have the right result within the precision F if $|q^n| < 10^{-F}$. The proceeding of `computeEtaViaFourierSeries` is now obvious. Again the practical performance of it for some sample input may be found in Figure 7.1 and Table 7.5, respectively.

In our application we have $\tau = \tau_Q$ with some reduced representative $Q = (a, b, c)$ of discriminant Δ. From $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ and $a \leq \sqrt{|\Delta|/3}$ it is easy to see that $|q| \leq e^{-\pi\sqrt{3}}$. Hence as we know e_n up to $e_{5000000}$, our algorithm `computeEtaViaFourierSeries` terminates successfully

Algorithm 7.6: `computeEtaViaProduct`(τ, F)

Input: A complex number $\tau \in \mathfrak{h}$.
A natural number F serving as floating point precision.
Output: The value $\eta(\tau)$ within the floating precision F.

$q \leftarrow e^{2\pi i\tau}$; $q_n \leftarrow q$; //q_n stores the current value q^n
$\eta \leftarrow 1$; //η stores the current approximation of $\eta(\tau)/q^{\frac{1}{24}}$
for $k = 0$ to 4 **do**
  $\eta \leftarrow \eta \cdot (1 - q_n)$; $q_n \leftarrow q_n \cdot q$;
  $\eta_l[k] \leftarrow \eta$; //$\eta_l$ stores the previous values of η
while true **do**
  $\eta \leftarrow \eta \cdot (1 - q_n)$; $q_n \leftarrow q_n \cdot q$; $\eta_b \leftarrow$ `true`;
  for $k = 0$ to 4 **do**
    if $|\eta - \eta_l[k]| > 10^{-F}$ **then**
      $\eta_b \leftarrow$ `false`; break;
  if $\eta_b =$ `true` **then**
    return($q^{\frac{1}{24}} \cdot \eta$);
  for $k = 1$ to 4 **do**
    $\eta_l[k-1] \leftarrow \eta_l[k]$;
  $\eta_l[4] \leftarrow \eta$;

for all F with $F \leq \frac{5000000 \cdot \pi\sqrt{3}}{\log 10} = 11815846$. However, this floating point precision is too large to be of practical relevance.

Finally, we explain our third algorithm `computeEtaViaEulerSum`(τ, F). This algorithm uses the Euler sum of Equation (7.22) to represent the η-function. The Euler sum may be written as follows, too:

$$q^{\frac{1}{24}} \sum_{n \in \mathbb{Z}} (-1)^n q^{\frac{3n^2+n}{2}} = q^{\frac{1}{24}} \left(1 + \sum_{n=1}^{\infty} (-1)^n (q^{\frac{3n^2-n}{2}} + q^{\frac{3n^2+n}{2}}) \right). \tag{7.35}$$

The basic idea of `computeEtaViaEulerSum` is to split the sum of the right side of (7.35) in two partial sums. We introduce new counting variables n_- and n_+ for either partial sum. In addition we set $N_-(n_-) = \frac{3n_-^2 - n_-}{2}$ and $N_+(n_+) = \frac{3n_+^2 + n_+}{2}$, respectively. Thus N_- and N_+ are the exponents of q in either partial sum, respectively. The difference of two exponents is $N_-(n_- + 1) - N_-(n_-) = 3n_- + 1$ and $N_+(n_+ + 1) - N_+(n_+) = 3n_+ + 2$, respectively. It is easy to see that we always have $N_- \neq N_+$.

In `computeEtaViaEulerSum` we store the current approximation of $\eta(\tau)/q^{\frac{1}{24}}$ in the variable η, that is

$$\eta = 1 + \sum_{n=1}^{n_- - 1} (-1)^n q^{\frac{3n^2-n}{2}} + \sum_{n=1}^{n_+ - 1} (-1)^n q^{\frac{3n^2+n}{2}} \ . \tag{7.36}$$

In addition, we take care that in (7.36) the exponents N_- and N_+ are adjacent, that is we ensure either $N_-(n_- - 1) < N_+(n_+)$ or $N_+(n_+ - 1) < N_-(n_-)$.

We first initialize $n_- = 2$ and $n_+ = 2$ and hence $N_- = 5$ and $N_+ = 7$. Furthermore, we initialize $\eta = 1 - q - q^2$. In addition, we introduce a variable l which holds the previously used exponent of q; thus we have to initialize $l = 2$. Next, in the while-loop we use an integer s to

Algorithm 7.7: `computeEtaViaFourierSeries`(τ, F)

Input: A complex number $\tau \in \mathfrak{h}$.
A natural number F serving as floating point precision.
Output: The value $\eta(\tau)$ within the floating precision F.

```
q ← e^{2πiτ}; q_n ← 1; // q_n stores the current value q^n
η ← e_0; η stores the current approximation of η(τ)/q^{1/24}
n ← 1;
while true do
  l ← n − 1;
  while e_n = 0 do
    n ← n + 1;
  q_n ← q_n · q^{n−l};
  η ← η + e_n · q_n;
  if |q_n| < 10^{−F} then
    return( q^{1/24} · η );
  n ← n + 1;
```

keep track of the sign of q_n; we first set $s = 1$. Now, let m denote the minimum of $\{N_-, N_+\}$. The algorithm passes through the corresponding assignments. In either loop we compute $q_m = q_l \cdot q^{m-l}$. If m is odd, the sign of q_m is -1; thus we set $s = -1$. Then we adapt either the value of N_- or N_+. Finally, we compute the new value of η. Again, as the Euler sum is sparse we are convinced to have the correct result if $|q_n| < 10^{-F}$. Again running times of this algorithm are given in Figure 7.1 and in Table 7.5.

We next discuss the practical performance of our three algorithms. We only test the performance in the context of our generating algorithm `cryptoCurve`. Thus let Δ be a fundamental discriminant of class number h. In addition, let $Q = (a, b, c)$ be a reduced representative of discriminant Δ. As usual we set $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ and $q = e^{2\pi i \tau_Q}$. Obviously $|q| = e^{-\pi\sqrt{|\Delta|}/a}$; thus the running time to compute the value $\eta(\tau_Q)$ mainly depends on a. This fact was already stated in Section 2.2.3.

In Figure 7.1 we plot the running time of either algorithm as a function of a, where a is the first entry of a reduced representative (a, b, c) of discriminant $\Delta = -21311$. We remark that Δ is the maximal fundamental discriminant of class number 200. In our computation we use the precision $F = 2163$, which comes from Formula (2.23). Figure 7.1 indicates that `computeEtaViaEulerSum`(τ_Q, F) is the most efficient of our algorithms to compute $\eta(\tau_Q)$ for a high precision F, i.e. $F \geq 2000$. In addition, the run time benefit of `computeEtaViaEulerSum` seems to become more definitely with growing a, hence with growing $|q|$, which is the important case for computing ring class polynomials. However, we tested our algorithms for low precisions and got similar results.

In addition, we tested our algorithms for various imaginary quadratic discriminants. We present some of the practical data in Table 7.5. The result is the same as in the case $\Delta = -21311$. Hence, it affirms our claims. All discriminants in Table 7.5 are maximal for a given class number.

In the context of algorithm `cryptoCurve` we have to compute the values $\eta(\tau_Q)$ where $Q = (a, b, c)$ is a reduced representative of discriminant Δ with $b \geq 0$. The reason is that if $g(\tau_Q)$

Algorithm 7.8: `computeEtaViaEulerSum`(τ, F)

Input: A complex number $\tau \in \mathfrak{h}$.
A natural number F serving as floating point precision.
Output: The value $\eta(\tau)$ within the floating point precision F.

```
1: n_- ← 2; n_+ ← 2;
2: N_- ← 5; N_+ ← 7; //to store the exponents of the partial sums
3: l ← 2; //to store the previous exponent
4: q ← e^{2πiτ}; q_n ← q^2; //q_n stores the current value q^n
5: η ← 1 − q − q_n; //η stores the current approximation of η(τ)/q^{1/24}
6: while true do
7:    s ← 1; //s stores the sign
8:    if N_- < N_+ then
9:       q_n ← q_n · q^{N_- − l};
10:      if 2 ∤ n_- then
11:         s ← −1;
12:      l ← N_-;
13:      N_- ← N_- + 3n_- + 1; n_- ← n_- + 1;
14:   else
15:      q_n ← q_n · q^{N_+ − l};
16:      if 2 ∤ n_+ then
17:         s ← −1;
18:      l ← N_+;
19:      N_+ ← N_+ + 3n_+ + 2; n_+ ← n_+ + 1;
20:   η ← η + s · q_n;
21:   if |q_n| < 10^{−F} then
22:      return( q^{1/24} · η );
```

is a class invariant, $g(\tau_{-Q}) = \overline{g(\tau_Q)}$ is a class invariant, too. In all, there are $h_p(\Delta)$ such reduced representatives. Thus in Table 7.5 we list the accumulated running time to compute $\eta(\tau_Q)$ for all $h_p(\Delta)$ reduced representatives with $b \geq 0$.

7.2.2 Efficient Computation of the Weber function $\mathfrak{f}_2$

In this section we compare different representations of the Weber function $\mathfrak{f}_2$ to compute the value $\mathfrak{f}_2(\tau)$ in practice, if $\tau \in \mathfrak{h}$ is given. We use three alternatives to represent $\mathfrak{f}_2(\tau)$:

- First, we make use of the defining equation (2.17), that is $\mathfrak{f}_2(\tau) = \sqrt{2} q^{\frac{1}{24}} \prod_{n=1}^{\infty}(1+q^n)$.
- Second, we compute $\mathfrak{f}_2(\tau)$ by evaluating its Fourier series $\sqrt{2} q^{\frac{1}{24}} \sum_{n=0}^{\infty} f_{2,n} q^n$ from Section 7.1.4.
- Finally, we use the relation $\mathfrak{f}_2(\tau) = \sqrt{2} \cdot \frac{\eta(2\tau)}{\eta(\tau)}$ ([Cox89], Equation (12.14), p. 256).

The corresponding algorithms are `computeF2ViaProduct`, `computeF2ViaFourierSeries`, and `computeF2ViaEta`, respectively. It turns out that using `computeF2ViaEta` is the most efficient approach in practice. In addition, we show that the running times of `computeF2ViaEta` and `computeEtaViaEulerSum` are approximately the same.

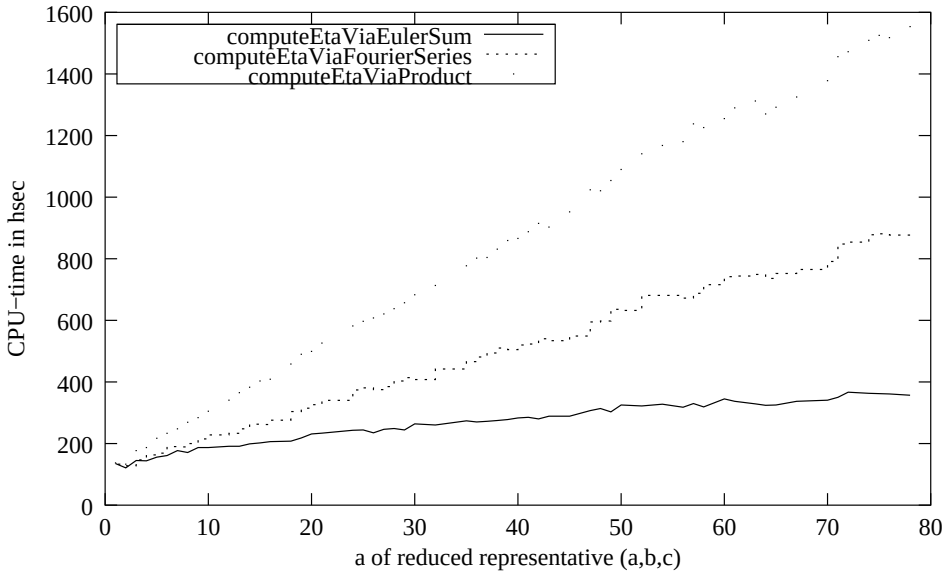


Figure 7.1: Running times on the SUN UltraSPARC-IIi of our three alternatives to compute $\eta(\tau_{(a,b,c)})$ for $(a,b,c) \in C(-21311)$. We use the precision $F = 2163$.

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_e in s	t_f in s	t_p in s	t_e/t_f	t_e/t_p	t_f/t_p
-21311	200	101	2163	276.53	525.96	875.17	0.525762	0.315972	0.600980
-30551	250	126	2776	539.09	1082.06	1830.59	0.498207	0.294489	0.591099
-34271	300	151	3206	858.73	1807.50	3072.44	0.475092	0.279494	0.588294
-47759	350	176	3883	1656.29	3723.35	6442.08	0.444838	0.257104	0.577973
-67031	400	201	4698	2711.40	6404.25	11114.49	0.423375	0.243951	0.576207
-75599	450	226	5193	3549.77	8600.98	14971.23	0.412716	0.237106	0.574500
-96599	500	251	5971	5226.91	13056.94	22763.12	0.400316	0.229621	0.573600

Table 7.5: Total running times on the SUN UltraSPARC-IIi of our three algorithms to compute all values $\eta(\tau_{(a,b,c)})$ for $(a,b,c) \in C(\Delta)$ with $b \geq 0$. t_e denotes the running time of `computeEtaViaEulerSum`, t_f the running time of `computeEtaViaFourierSeries`, and t_p the running time of `computeEtaViaProduct`, respectively.

As in the previous section, the input of either algorithm is a complex number $\tau \in \mathfrak{h}$ and a floating point precision F. Each algorithm returns the value $\mathfrak{f}_2(\tau)$ within the precision F, that is either algorithm returns a complex number $\mathfrak{f}_{2,\tau}$ with $|\mathfrak{f}_{2,\tau} - \mathfrak{f}_2(\tau)| < 10^{-F}$.

We first turn to our algorithm `computeF2ViaProduct`(τ, F). It is a straightforward consequence of the defining equation (2.17) of the Weber function $\mathfrak{f}_2$. In addition, as the infinite product in (2.17) is up to sign of q^n the same as in the defining equation (2.14) of Dedekind's η-function, `computeF2ViaProduct` is very similar to algorithm `computeEtaViaProduct`; this algorithm was explained in Section 7.2.1. Thus we skip a detailed description and listing of `computeF2ViaProduct`.

Next, we describe algorithm `computeF2ViaFourierSeries`. However, again this algorithm is very similar to algorithm `computeEtaViaFourierSeries`. Thus we leave out a further description of this algorithm, too.

Finally, we explain `computeF2ViaEta`. This algorithm bases on `computeEtaViaEulerSum`. `computeF2ViaEta` makes use of the relation $\mathfrak{f}_2(\tau_Q) = \sqrt{2}\frac{\eta(2\tau_Q)}{\eta(\tau_Q)}$. The main observation is that we can compute the numerator and denominator of this fraction simultaneously; this is an obvious consequence of $q(2\tau) = q^2$. We get algorithm `computeF2ViaEta`(τ, F) from `computeEtaViaEulerSum`(τ, F) if we change the following lines (η_2 stores the current approximation of $\eta(2\tau)/q^{\frac{1}{12}}$):

Line	`computeEtaViaEulerSum`	`computeF2ViaEta`
5	$\eta \leftarrow 1 - q - q_n$;	$\eta \leftarrow 1 - q - q_n$; $\eta_2 \leftarrow 1 - q^2 - q_n^2$;
20	$\eta \leftarrow \eta + s \cdot q_n$;	$\eta \leftarrow \eta + s \cdot q_n$; $\eta_2 \leftarrow \eta_2 + s \cdot q_n^2$;
22	return($q^{\frac{1}{24}} \cdot \eta$);	return($\sqrt{2}q^{\frac{1}{24}} \cdot \frac{\eta_2}{\eta}$);

Obviously `computeF2ViaEta` returns the value $\mathfrak{f}_2(\tau)$ within the precision F. We next discuss the practical performance of our three algorithms. As in the previous section we only test the performance in the context of our main algorithm `cryptoCurve`. Hence we only present running times of our algorithms for input of the form (τ_Q, F), where as usual $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ for a reduced representative $Q = (a, b, c)$ of fundamental discriminant Δ.

Again, we first compare the running times of our three algorithms of this section for the reduced representatives of $C(-21311)$; the running times of either algorithm is plotted in Figure 7.2 as a function of a. Again we use the floating point precision $F = 2163$ in our computation; this precision comes from Formula (2.23). It is obvious from Figure 7.2 that the running times of either algorithm of this section is approximately the same as the running time of the corresponding algorithm of Section 7.2.1 to compute $\eta(\tau_Q)$.

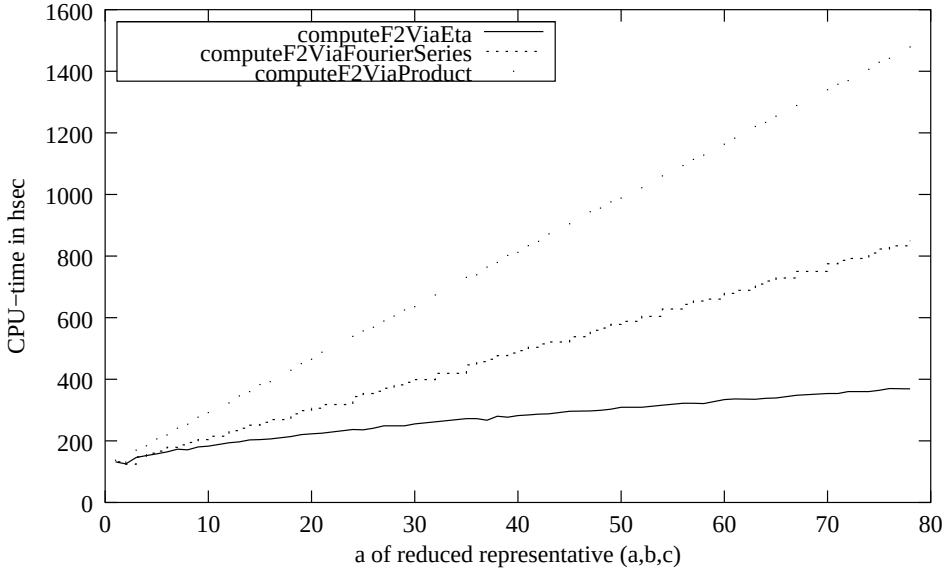


Figure 7.2: Running times on the SUN UltraSPARC-IIi of our three alternatives to compute $\mathfrak{f}_2(\tau_{(a,b,c)})$ for $(a, b, c) \in C(-21311)$. We use the precision $F = 2163$.

Furthermore, Figure 7.2 indicates that `computeF2ViaEta`(τ, F) is the most efficient of our algorithms to compute $\mathfrak{f}_2(\tau_Q)$ for high precision F, i.e. $F \geq 2000$. However, corresponding to our experiments the same holds for low precisions. In addition, as in the case of Dedekind's η-function, the run time benefit of `computeF2ViaEta` seems to become more definitely with growing a.

In addition, we tested our algorithms of this section for various further imaginary quadratic discriminants. Some of the practical data is given in Table 7.6. The result of the further discriminants is the same as in the case $\Delta = -21311$. Hence, again the additional data affirms our claims. All discriminants in Table 7.6 are maximal for a given class number.

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_η in s	t_f in s	t_p in s	t_e/t_f	t_e/t_p	t_f/t_p
-21311	200	101	2163	276.75	497.26	819.63	0.556549	0.337652	0.606688
-30551	250	126	2776	570.48	1090.39	1823.69	0.523188	0.312816	0.597903
-34271	300	151	3206	893.68	1784.12	2994.91	0.500908	0.298399	0.595711
-47759	350	176	3883	1743.68	3723.75	6387.12	0.468259	0.272999	0.583009
-67031	400	201	4698	2876.84	6455.77	11110.81	0.445623	0.258922	0.581035
-75599	450	226	5193	3751.35	8644.82	14928.11	0.433941	0.251294	0.579096
-96599	500	251	5971	5488.81	13046.69	22581.76	0.420705	0.243063	0.577753

Table 7.6: Total running times on the SUN UltraSPARC-IIi of our three algorithms to compute all values $\mathfrak{f}_2(\tau_{(a,b,c)})$ for $(a,b,c) \in C(\Delta)$ with $b \geq 0$. t_η denotes the running time of `computeF2ViaEta`, t_f the running time of `computeF2ViaFourierSeries`, and t_p the running time of `computeF2ViaProduct`.

7.2.3 Efficient Computation of the Weber function $\mathfrak{f}_1$

In this section we compare different representations of the Weber function $\mathfrak{f}_1$ to compute the value $\mathfrak{f}_1(\tau)$ in practice. We implement the following three representations of $\mathfrak{f}_1(\tau)$:

- First, we make use of the defining equation (2.16), that is $\mathfrak{f}_1(\tau) = q^{-\frac{1}{48}} \prod_{n=1}^{\infty}(1-q^{n-\frac{1}{2}})$.
- Second, we compute $\mathfrak{f}_1(\tau)$ by evaluating its Fourier series $q^{-\frac{1}{48}} \sum_{n=0}^{\infty} f_{1,n} q^{n/2}$ from Section 7.1.5.
- Finally, we use the well-known relation $\mathfrak{f}_1(\tau) = \frac{\eta(\tau/2)}{\eta(\tau)}$ ([Cox89], Equation (12.14), p. 256).

The corresponding algorithms are `computeF1ViaProduct`, `computeF1ViaFourierSeries`, and `computeF1ViaEta`, respectively. We show that using `computeF1ViaEta` is the most efficient approach in practice. In addition, we show that `computeF2ViaEta`(τ, F) is faster than `computeF1ViaEta`(τ, F).

As in the previous sections, the input of either algorithm is a complex number $\tau \in \mathfrak{h}$ and a floating point precision F. Each algorithm returns the value $\mathfrak{f}_1(\tau)$ within the precision F, that is either algorithm returns a complex number $\mathfrak{f}_{1,\tau}$ with $|\mathfrak{f}_{1,\tau} - \mathfrak{f}_1(\tau)| < 10^{-F}$.

The algorithms `computeF1ViaProduct`(τ, F) and `computeF1ViaFourierSeries`(τ, F) are analogous to the algorithms to compute $\eta(\tau)$ and $\mathfrak{f}_2(\tau)$, respectively. Hence they are straightforward, and their implementation is obvious. Thus we leave out a detailed description.

However, we explain `computeF1ViaEta` in detail. As mentioned above it makes use of the relation

$$\begin{aligned}\mathfrak{f}_1(\tau) &= \frac{\eta(\frac{\tau}{2})}{\eta(\tau)} \\ &= q^{-\frac{1}{48}} \cdot \frac{1+\sum_{n=1}^{\infty}(-1)^n(q^{\frac{3n^2-n}{4}}+q^{\frac{3n^2+n}{4}})}{1+\sum_{n=1}^{\infty}(-1)^n(q^{\frac{3n^2-n}{2}}+q^{\frac{3n^2+n}{2}})} \quad . \end{aligned} \tag{7.37}$$

It is obvious from Equation (7.37) that the q-powers of the denominator are the squares of the q-powers of the numerator, respectively. Hence we can compute numerator and denominator of Equation (7.37) simultaneously.

Our algorithm `computeF1ViaEta` mainly bases on algorithm `computeEtaViaEulerSum`. However, we make use of a variable $q_{1/2} = e^{\pi i \tau}$; the reason is that in the numerator of Equation (7.37) we have to know powers of $q^{n/2}$. Furthermore, we introduce a variable $\eta_{1/2}$ which stores the current approximation of $\eta(\tau/2)/e^{\frac{\pi i \tau}{24}}$. The further implementation of `computeF1ViaEta` is the same as of `computeEtaViaEulerSum`. Thus we refer to Section 7.2.1 for details.

It is easy to see that `computeF1ViaEta` returns the correct result $\mathfrak{f}_1(\tau)$ within the precision F. We next turn to the practical performance of our three algorithms. As in the previous sections we only test the performance in the context of algorithm `cryptoCurve`. Thus, again we only present timings of our algorithms for input of the form (τ_Q, F), where as usual $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ for a reduced representative $Q = (a, b, c)$ of fundamental discriminant Δ.

Again, we first compare the running times of our three algorithms of this section for the reduced representatives of $C(-21311)$; the running times of either algorithm is plotted in Figure 7.3 as a function of a. Again we use the floating point precision $F = 2163$ in our computation; this precision comes from Formula (2.23). It is obvious from Figure 7.3 that only the timings of `computeF1ViaProduct` and `computeF2ViaProduct` are approximately the same. The reason is that the products in (2.16) and (2.17) of $\mathfrak{f}_1$ and $\mathfrak{f}_2$ obviously have the same convergence behavior, respectively.

However, in case of `computeF1ViaFourierSeries` and `computeF1ViaEta` the corresponding algorithms to compute $\mathfrak{f}_2(\tau_Q)$ are faster, respectively; the reason is that in either case the convergence of the corresponding series of $\mathfrak{f}_2(\tau_Q)$ is obviously better, as we make use of q instead of $q_{1/2}$ in this case. Hence, in case of $\mathfrak{f}_2$ we achieve the truncation condition $|q^n| < 10^{-F}$ for smaller values $n \in \mathbb{N}$ than in case of $\mathfrak{f}_1$.

In addition, Figure 7.3 indicates that `computeF1ViaEta`(τ, F) is the most efficient of our algorithms to compute $\mathfrak{f}_1(\tau_Q)$ for high precision F, that is $F \geq 2000$. However, corresponding to our experiments the same holds for low precisions. In addition, as in the case of Dedekind's η-function and the Weber function $\mathfrak{f}_2$, the run time benefit of `computeF1ViaEta` seems to become more definitely with growing a.

Furthermore, we tested our algorithms of this section for various further imaginary quadratic discriminants. Some of the practical data is given in Table 7.7. The result of the further discriminants is the same as in the case $\Delta = -21311$. Hence, again the additional data

Algorithm 7.9: `computeF1ViaEta`(τ, F)

Input: A complex number $\tau \in \mathfrak{h}$.
A natural number F serving as floating point precision.
Output: The value $\mathfrak{f}_1(\tau)$ within the floating point precision F.

$n_- \leftarrow 2$; $n_+ \leftarrow 2$;
$N_- \leftarrow 5$; $N_+ \leftarrow 7$; //to store the exponents of the partial sums
$l \leftarrow 2$; //to store the previous exponent
$q_{1/2} \leftarrow e^{\pi i \tau}$; $q_{n,1/2} \leftarrow q_{1/2}^2$; //$q_{n,1/2}$ stores the current value $q_{1/2}^n$
$\eta_{1/2} \leftarrow 1 - q_{1/2} - q_{n,1/2}$; //$\eta_{1/2}$ stores the current approximation of $\eta(\tau/2)/e^{\frac{\pi i \tau}{24}}$
$\eta \leftarrow 1 - q_{1/2}^2 - q_{n,1/2}^2$; //$\eta$ stores the current approximation of $\eta(\tau)/e^{\frac{2\pi i \tau}{24}}$
while true **do**
  $s \leftarrow 1$; //s stores the sign
  if $N_- < N_+$ **then**
    $q_{n,1/2} \leftarrow q_{n,1/2} \cdot q_{1/2}^{N_- - l}$;
    if $2 \nmid n_-$ **then**
      $s \leftarrow -1$;
    $l \leftarrow N_-$;
    $N_- \leftarrow N_- + 3n_- + 1$; $n_- \leftarrow n_- + 1$;
  else
    $q_{n,1/2} \leftarrow q_{n,1/2} \cdot q_{1/2}^{N_+ - l}$;
    if $2 \nmid n_+$ **then**
      $s \leftarrow -1$;
    $l \leftarrow N_+$;
    $N_+ \leftarrow N_+ + 3n_+ + 2$; $n_+ \leftarrow n_+ + 1$;
  $\eta_{1/2} \leftarrow \eta_{1/2} + s \cdot q_{n,1/2}$; $\eta \leftarrow \eta + s \cdot q_{n,1/2}^2$;
  if $|q_{n,1/2}| < 10^{-F}$ **then**
    return($q_{1/2}^{-\frac{1}{24}} \cdot \frac{\eta_{1/2}}{\eta}$);

affirms our claims. Again we point out the run time benefit of `computeF2ViaEta` compared to `computeF1ViaEta`. All discriminants in Table 7.7 are maximal for a given class number.

7.2.4 Efficient Computation of the Weber function $\mathfrak{f}$

In this section we turn to our three approaches to compute $\mathfrak{f}(\tau)$. Our proceeding is similar to our investigation in the previous sections. We make use of the following three alternatives to represent $\mathfrak{f}(\tau)$:

- First, we make use of the defining equation (2.15), i.e. $\mathfrak{f}(\tau) = q^{-\frac{1}{48}} \prod_{n=1}^{\infty}(1 + q^{n-\frac{1}{2}})$.
- Second, we compute $\mathfrak{f}(\tau)$ by evaluating its Fourier series $q^{-\frac{1}{48}} \sum_{n=0}^{\infty} f_n q^{n/2}$ from Section 7.1.5.
- Finally, we use the well-known relation $\mathfrak{f}(\tau) = e^{-\frac{2\pi i}{48}} \frac{\eta((\tau+1)/2)}{\eta(\tau)}$ ([Cox89], Equation (12.14), p. 256).

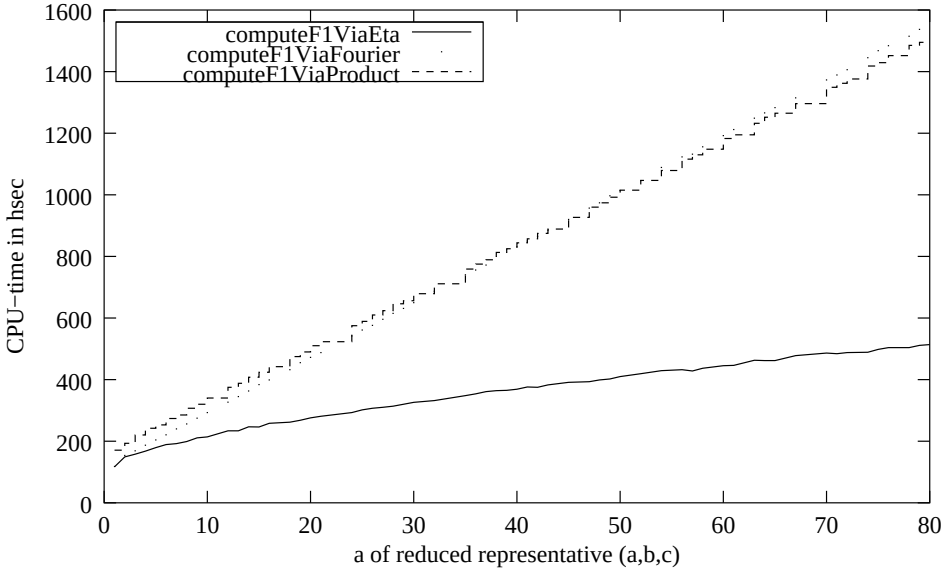


Figure 7.3: Running times on the SUN UltraSPARC-IIi of our three alternatives to compute $\mathfrak{f}_1(\tau_{(a,b,c)})$ for $(a,b,c) \in C(-21311)$. We use the precision $F = 2163$.

We call the corresponding algorithms `computeFViaProduct`, `computeFViaFourierSeries`, and `computeFViaEta`, respectively. We show that using `computeFViaEta` is the most efficient approach in practice. In addition, we show that `computeF2ViaEta`(τ, F) is faster than `computeFViaEta`(τ, F).

As in the previous sections, the input of either algorithm is a complex number $\tau \in \mathfrak{h}$ and a floating point precision F. Each algorithm returns the value $\mathfrak{f}(\tau)$ within the precision F, that is either algorithm returns a complex number $\mathfrak{f}_\tau$ with $|\mathfrak{f}_\tau - \mathfrak{f}(\tau)| < 10^{-F}$.

The algorithms `computeFViaProduct`(τ, F) and `computeFViaFourierSeries`(τ, F) are analogous to the algorithms to compute $\eta(\tau)$, $\mathfrak{f}_2(\tau)$, and $\mathfrak{f}_1(\tau)$, respectively. Hence they are straightforward, and their implementation is obvious. Thus we skip a detailed description.

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_η in s	t_f in s	t_p in s	t_e/t_f	t_e/t_p	t_f/t_p
-21311	200	101	2163	361.05	838.68	850.77	0.430497	0.424380	0.985789
-30551	250	126	2776	752.49	1879.07	1901.17	0.400458	0.395803	0.988375
-34271	300	151	3206	1169.49	3074.27	3091.10	0.380412	0.378341	0.994555
-47759	350	176	3883	2071.62	5805.93	5827.96	0.356811	0.355462	0.996219
-67031	400	201	4698	3780.04	11306.09	11304.36	0.334336	0.334387	1.0001
-75599	450	226	5193	4929.12	15195.27	15200.34	0.324385	0.324276	0.999666
-96599	500	251	5971	7134.08	23045.32	23001.43	0.309567	0.310158	1.00190

Table 7.7: Total running times on the SUN UltraSPARC-IIi of our three algorithms to compute all values $\mathfrak{f}_1(\tau_{(a,b,c)})$ for $(a,b,c) \in C(\Delta)$ with $b \geq 0$. t_η denotes the running time of `computeF1ViaEta`, t_f the running time of `computeF1ViaFourierSeries`, and t_p the running time of `computeF1ViaProduct`.

We next explain `computeFViaEta`. As mentioned above it makes use of the relation

$$\begin{aligned}\mathfrak{f}(\tau) &= e^{-\frac{2\pi i}{48}} \cdot \frac{\eta(\frac{\tau+1}{2})}{\eta(\tau)} \\ &= q^{-\frac{1}{48}} \cdot \frac{1+\sum_{n=1}^{\infty}(-1)^n(e^{2\pi i\cdot\frac{3n^2-n}{4}}q^{\frac{3n^2-n}{4}}+e^{2\pi i\cdot\frac{3n^2+n}{4}}q^{\frac{3n^2+n}{4}})}{1+\sum_{n=1}^{\infty}(-1)^n(q^{\frac{3n^2-n}{2}}+q^{\frac{3n^2+n}{2}})} \quad . \end{aligned} \tag{7.38}$$

As in case of the Weber function $\mathfrak{f}_1$ it is obvious from Equation (7.38) that the q-powers of the denominator are the squares of the q-powers of the numerator, respectively. Hence, again we can compute numerator and denominator of Equation (7.38) simultaneously.

However, we have to determine the factors $e^{2\pi i\cdot\frac{3n^2-n}{4}}$ and $e^{2\pi i\cdot\frac{3n^2+n}{4}}$, respectively. Obviously $3n^2-n$ is even for all $n \in \mathbb{N}$. We set $N := \frac{3n^2-n}{2}$ in what follows. Thus $e^{2\pi i\cdot\frac{3n^2-n}{4}} = 1$ if N is even, and $e^{2\pi i\cdot\frac{3n^2-n}{4}} = -1$ otherwise. The same is true for $e^{2\pi i\cdot\frac{3n^2+n}{4}}$. As in the previous section we set $q_{1/2} = e^{\pi i\tau}$. The further design of `computeFViaEta` resembles algorithm `computeF1ViaEta`. Thus we refer to this algorithm for details.

It is easy to see that `computeFViaEta` returns the value $\mathfrak{f}(\tau)$ within the precision F. We next discuss the practical performance of our three algorithms. As in the previous sections we only test the performance in the scope of our generating algorithm. Thus, again we only present timings of our algorithms for input of the form (τ_Q, F), where as usual $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ for a reduced representative $Q = (a, b, c)$ of fundamental discriminant Δ.

Again, we first compare the running times of our three algorithms of this section for the reduced representatives of $C(-21311)$; the running times of either algorithm is plotted in Figure 7.4 as a function of a. Again we use the floating point precision $F = 2163$ in our computation; this precision comes from Formula (2.23). It is obvious from Figure 7.4 that the run time of either algorithm is about the same as the timing of the corresponding algorithm to compute $\mathfrak{f}_1(\tau_Q)$. The reason is that obviously either representation of $\mathfrak{f}$ and $\mathfrak{f}_1$, respectively, has the same convergence behavior.

However, `computeFViaFourierSeries` and `computeFViaEta` are significantly slower than the corresponding algorithms to compute $\mathfrak{f}_2(\tau_Q)$, respectively; again the reason is that in either case the convergence of the corresponding series of $\mathfrak{f}_2(\tau_Q)$ is obviously better, as we make use of q instead of $q_{1/2}$ in this case.

In addition, Figure 7.4 indicates that `computeFViaEta`(τ, F) is the most efficient of our algorithms to compute $\mathfrak{f}(\tau_Q)$ for high precision F, that is $F \geq 2000$. However, corresponding to our experiments the same holds for low precisions. In addition, as in the case of Dedekind's η-function and the Weber functions $\mathfrak{f}_2$ and $\mathfrak{f}_1$, the run time benefit of `computeFViaEta` seems to become more definitely with growing a.

Furthermore, we tested our algorithms of this section for various further imaginary quadratic discriminants. Some of the practical data is given in Table 7.8. The result of the further discriminants is the same as in the case $\Delta = -21311$. Hence, again the additional data affirms our claims. Again we point out the run time benefit of `computeF2ViaEta` compared to `computeFViaEta`. All discriminants in Table 7.8 are maximal for a given class number.

Algorithm 7.10: `computeFViaEta`(τ, F)

Input: A complex number $\tau \in \mathfrak{h}$.
A natural number F serving as floating point precision.
Output: The value $\mathfrak{f}(\tau)$ within the floating point precision F.

$n_- \leftarrow 2$; $n_+ \leftarrow 2$; $N_- \leftarrow 5$; $N_+ \leftarrow 7$; // to store the exponents of the partial sums
$l \leftarrow 2$; // to store the previous exponent
$q_{1/2} \leftarrow e^{\pi i \tau}$; $q_{n,1/2} \leftarrow q_{1/2}^2$; // $q_{n,1/2}$ stores the current value $q_{1/2}^n$
$\eta_{1/2} \leftarrow 1 + q_{1/2} - q_{n,1/2}$; // $\eta_{1/2}$ stores the current approximation of $\eta((\tau+1)/2)/e^{2\pi i(\tau+1)/48}$
$\eta \leftarrow 1 - q_{1/2}^2 - q_{n,1/2}^2$; // η stores the current approximation of $\eta(\tau)/e^{2\pi i \tau/24}$
while true **do**
  $s_1 \leftarrow 1$; $s_2 \leftarrow 1$; //s_1 stores the sign in the numerator, s_2 stores the sign in the denominator
  if $N_- < N_+$ **then**
    $q_{n,1/2} \leftarrow q_{n,1/2} \cdot q_{1/2}^{N_- - l}$;
    if $2 \nmid n_-$ **then**
      $s_2 \leftarrow -1$;
      if $2 \mid N_-$ **then**
        $s_1 \leftarrow -1$;
    else if $2 \mid n_-$ AND $2 \nmid N_-$ **then**
      $s_1 \leftarrow -1$;
    $l \leftarrow N_-$; $N_- \leftarrow N_- + 3n_- + 1$; $n_- \leftarrow n_- + 1$;
  else
    $q_{n,1/2} \leftarrow q_{n,1/2} \cdot q_{1/2}^{N_+ - l}$;
    if $2 \nmid n_+$ **then**
      $s_2 \leftarrow -1$;
      if $2 \mid N_+$ **then**
        $s_1 \leftarrow -1$;
    else if $2 \mid n_+$ AND $2 \nmid N_+$ **then**
      $s_1 \leftarrow -1$;
    $l \leftarrow N_+$; $N_+ \leftarrow N_+ + 3n_+ + 2$; $n_+ \leftarrow n_+ + 1$;
  $\eta_{1/2} \leftarrow \eta_{1/2} + s_1 \cdot q_{n,1/2}$; $\eta \leftarrow \eta + s_2 \cdot q_{n,1/2}^2$;
  if $|q_{n,1/2}| < 10^{-F}$ **then**
    return($q_{1/2}^{-\frac{1}{24}} \cdot \frac{\eta_{1/2}}{\eta}$);

7.2.5 Efficient Computation of the functions γ_2 and j

In this section we describe efficient algorithms to compute $\gamma_2(\tau)$ and $j(\tau)$, respectively. The functions are related by $j = \gamma_2^3$. Thus we first investigate efficient algorithms to evaluate $\gamma_2(\tau)$. We make use of two different representations:

- We first represent $\gamma_2(\tau)$ by its Fourier series $q^{-\frac{1}{3}} \cdot \sum_{n=0}^{\infty} g_n q^n$ of Section 7.1.2.
- We make use of Equation (2.20), that is

$$\gamma_2(\tau) = \frac{\mathfrak{f}_2^{24}(\tau) + 16}{\mathfrak{f}_2^8(\tau)} \ . \tag{7.39}$$

Our algorithms `computeGamma2ViaFourierSeries`(τ, F) and `computeGamma2ViaEta`(τ, F) implement these representations, respectively. However, as a result of the previous sections

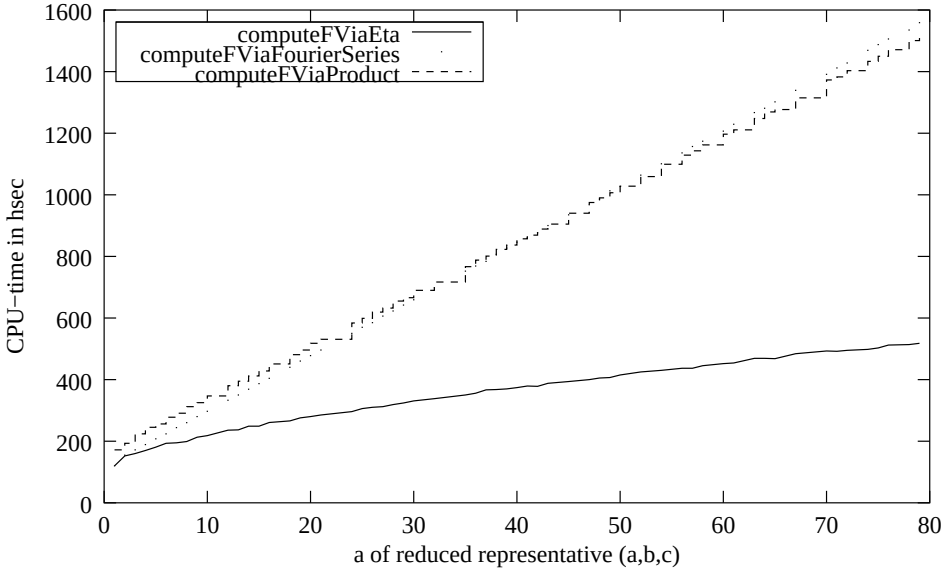


Figure 7.4: Running times on the SUN UltraSPARC-IIi of our three alternatives to compute $\mathfrak{f}(\tau_{(a,b,c)})$ for $(a,b,c) \in C(-21311)$. We use the precision $F = 2163$.

we have that `computeF2ViaEta`(τ, F) turns out to be faster than both `computeFViaEta`(τ, F) and `computeF1ViaEta`(τ, F). For this reason we choose the Weber function $\mathfrak{f}_2$ in Equation (2.20).

Input of both algorithms is again a complex number $\tau \in \mathfrak{h}$ and a floating point precision F; both algorithms return the value $\gamma_2(\tau)$ within the precision F. The idea and the implementation of `computeGamma2ViaFourierSeries` is straightforward; we refer to the similar algorithm `computeEtaViaFourierSeries` of Section 7.2.1 for details. In addition, as we make use of (7.39), the same is true for `computeGamma2ViaEta`. However, we point out that in Equation (7.39) we use algorithm `computeF2ViaEta`(τ, F) to compute $\mathfrak{f}_2(\tau)$.

We next discuss the practical performance of our two algorithms. Again we only test the performance in the framework of our generating algorithm. Thus, we only present timings of our algorithms for input of the form (τ_Q, F), where as usual $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ for a reduced representative $Q = (a,b,c)$ of fundamental discriminant Δ.

As before, we first compare the running times of our algorithms of this section for the reduced representatives of $C(-21311)$; the running time of either algorithm is plotted in Figure 7.5 as a function of a. Again we use the floating point precision $F = 2163$ in our computation; this precision comes from Formula (2.23). It is obvious from Figures 7.2 and 7.5 that the run time of `computeGamma2ViaEta` is the same as the running time of `computeF2ViaEta`. However, the reason is that once we know $\mathfrak{f}_2(\tau)$ we can neglect the computational overhead of computing $\gamma_2(\tau)$ using Equation (7.39).

In addition, Figure 7.5 indicates that `computeGamma2ViaEta`(τ, F) is more efficient in practice to compute $\gamma_2(\tau_Q)$ for high precision F, that is $F \geq 2000$. However, corresponding to our experiments the same holds for low precisions. Again, the run time benefit of `computeGamma2ViaEta` seems to become more definitely with growing a.

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_η in s	t_f in s	t_p in s	t_e/t_f	t_e/t_p	t_f/t_p
-21311	200	101	2163	365.87	851.07	862.82	0.429894	0.424039	0.986381
-30551	250	126	2776	764.95	1914.91	1934.84	0.399470	0.395355	0.989699
-34271	300	151	3206	1182.52	3101.45	3120.72	0.381279	0.378925	0.993825
-47759	350	176	3883	2069.72	5802.45	5823.18	0.35669	0.355427	0.996440
-67031	400	201	4698	3776.43	11287.38	11291.61	0.334571	0.334445	0.999625
-75599	450	226	5193	4928.73	15189.38	15191.03	0.324485	0.324450	0.999891
-96599	500	251	5971	7134.83	23055.32	23006.41	0.309465	0.310123	1.00212

Table 7.8: Total running times on the SUN UltraSPARC-IIi of our three algorithms to compute all values $\mathfrak{f}(\tau_{(a,b,c)})$ for $(a,b,c) \in C(\Delta)$ with $b \geq 0$. t_η denotes the running time of `computeFViaEta`, t_f the running time of `computeFViaFourierSeries`, and t_p the running time of `computeFViaProduct`.

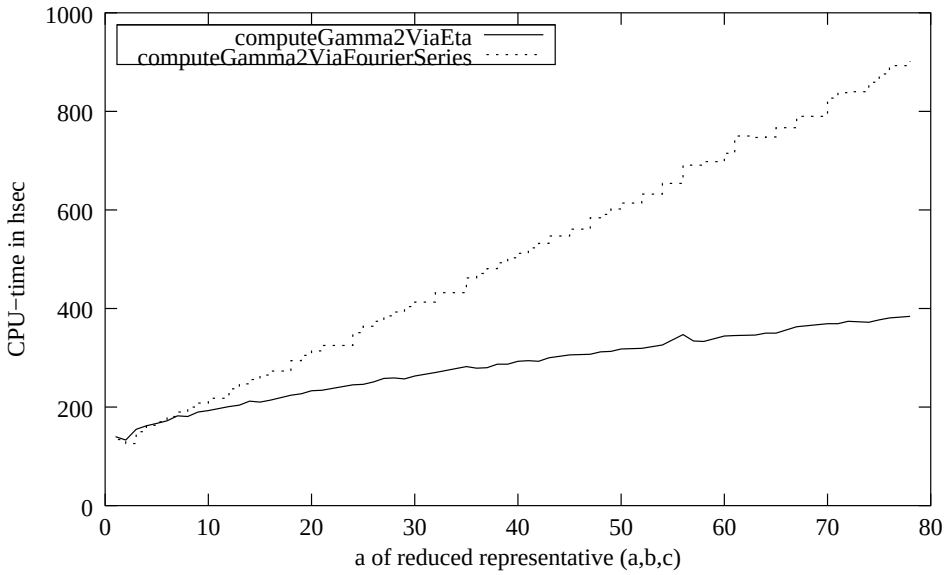


Figure 7.5: Running times on the SUN UltraSPARC-IIi of our two alternatives to compute $\gamma_2(\tau_{(a,b,c)})$ for $(a,b,c) \in C(-21311)$. We use the precision $F = 2163$.

Furthermore, we tested our algorithms to evaluate $\gamma_2(\tau)$ for various further imaginary quadratic discriminants. Some of the practical data is given in Table 7.9. The result of the further discriminants is the same as in the case $\Delta = -21311$. Again the additional data affirms our claims. Once more we point out that the run time of `computeF2ViaEta` and `computeGamma2ViaEta` are about the same. All discriminants in Table 7.9 are maximal for a given class number.

Finally, we turn to the computation of singular moduli, that is to the computation of $j(\tau_Q)$ if $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$. As in the case of the function γ_2 we evaluate two approaches:

- We first represent $j(\tau)$ by its Fourier series $\sum_{n=-1}^{\infty} c_n q^n$ of Section 7.1.1.
- We make use of the defining equation

$$j(\tau) = \gamma_2^3(\tau) \; . \tag{7.40}$$

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_η in s	t_f in s	t_η/t_f
-21311	200	101	2163	286.73	520.00	0.55140385
-30551	250	126	2776	606.18	1173.19	0.5166938
-34271	300	151	3206	957.56	1933.66	0.49520598
-47759	350	176	3883	1846.71	3966.54	0.46557201
-67031	400	201	4698	2909.43	6603.05	0.44061911
-75599	450	226	5193	3860.13	9021.52	0.42788022
-96599	500	251	5971	5685.60	13694.79	0.41516518

Table 7.9: Total running times on the SUN UltraSPARC-IIi of our two algorithms to compute all values $\gamma_2(\tau_{(a,b,c)})$ for $(a,b,c) \in C(\Delta)$ with $b \geq 0$. t_η denotes the running time of `computeGamma2ViaEta` and t_f the running time of `computeGamma2ViaFourierSeries`.

We call the algorithms `computeJViaFourierSeries`(τ, F) and `computeJViaEta`(τ, F), respectively. The latter one uses algorithm `computeGamma2ViaEta`(τ, F) to compute $\gamma_2(\tau)$ within the precision F. We show below that the practical performance of `computeJViaEta` is superior. In addition, we give evidence that the run time of `computeJViaEta` and `computeGamma2ViaEta` is about the same.

As usual the input of either algorithm is a complex number $\tau \in \mathfrak{h}$ and a floating point precision F; both algorithms return the value $j(\tau)$ within the precision F. As above the idea and the implementation of `computeJViaFourierSeries` is straightforward; again we refer to the similar algorithm `computeEtaViaFourierSeries` of Section 7.2.1 for details. In addition, as we make use of (7.40) the same holds for `computeJViaEta`. Once more we remark that in Equation (7.40) we use algorithm `computeGamma2ViaEta`(τ, F) to compute $\gamma_2(\tau)$.

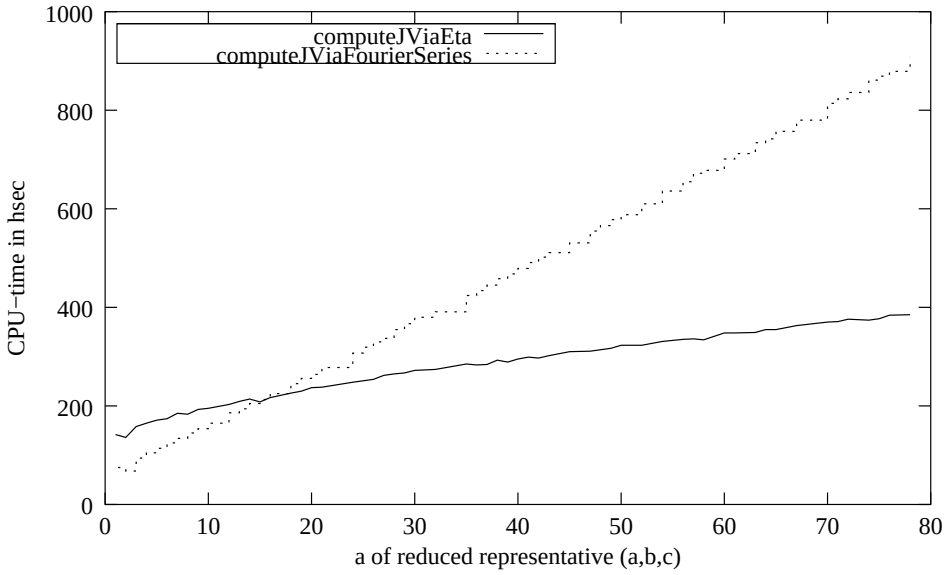


Figure 7.6: Running times on the SUN UltraSPARC-IIi of our two alternatives to compute $j(\tau_{(a,b,c)})$ for $(a,b,c) \in C(-21311)$. We use the precision $F = 2163$.

We turn to practical run times of our two algorithms. Again we only test the performance with regard to our generating algorithm. Thus, we only present timings of our algorithms for input of the form (τ_Q, F), where as usual $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ for a reduced representative $Q = (a, b, c)$ of fundamental discriminant Δ.

As before, we first compare the running times of our algorithms of this section for the reduced representatives of $C(-21311)$; the running time of either algorithm is plotted in Figure 7.6 as a function of a. Again we use the floating point precision $F = 2163$ in our computation; this precision comes from Formula (2.23). It is obvious from Figures 7.5 and 7.6 that the run time of `computeJViaEta` is approximately the same as the running time of `computeGamma2ViaEta`. The reason is that once we know $\gamma_2(\tau)$ we can neglect the computational overhead to compute $j(\tau)$ using Equation (7.40).

In addition, Figure 7.6 indicates that `computeJViaEta`(τ, F) is more efficient in practice to compute $j(\tau_Q)$ for high precision F, that is $F \geq 2000$. However, corresponding to our experiments the same holds for low precisions. Again, the run time benefit of `computeJViaEta` seems to become more definitely with growing a.

Furthermore, we tested our algorithms to evaluate $j(\tau)$ for various further imaginary quadratic discriminants. Some of the practical data is given in Table 7.10. The result of the further discriminants is the same as in the case $\Delta = -21311$. Again the additional data affirms our claims. Once more we point out that the run time of `computeJViaEta` and `computeGamma2ViaEta` are about the same. All discriminants in Table 7.10 are maximal for a given class number.

Δ	$h(\Delta)$	$h_p(\Delta)$	F	t_η in s	t_f in s	t_η/t_f
-21311	200	101	2163	290.10	486.86	0.59585918
-30551	250	126	2776	609.02	1068.57	0.55590181
-34271	300	151	3206	959.46	1765.62	0.52981955
-47759	350	176	3883	1856.78	3663.97	0.49366671
-67031	400	201	4698	2914.84	6235.32	0.46634976
-75599	450	226	5193	3884.89	8371.53	0.45307011
-96599	500	251	5971	5691.53	12827.73	0.43838855

Table 7.10: Total running times on the SUN UltraSPARC-IIi of the two alternatives to compute all values $j(\tau_{(a,b,c)})$ for $(a, b, c) \in C(\Delta)$ with $b \geq 0$. t_η denotes the running time of `computeJViaEta`, t_f the running time of `computeJViaFourierSeries`.

We close this section with pointing to a further advantage of algorithm `computeJViaEta`: It is independent of any precomputation and does not make use of coefficients which have to be stored. This is an obvious consequence of the same property of algorithm `computeF2ViaEta`, which is invoked by algorithm `computeGamma2ViaEta`.

7.3 The Algorithm `computeClassInvariants`

In this section we describe our algorithm `computeClassInvariants`(Δ, h, R). This algorithm is a subalgorithm of our main generating algorithm `cryptoCurve`. At the end of this section

we show that its bit-complexity is at most $O(h^6 \log h)$. The input is an imaginary quadratic discriminant Δ, its class number h, and an array R storing the reduced representatives of discriminant Δ. The algorithm returns an array I holding the roots of a generating polynomial of the ring class field $L/\mathbb{Q}(\sqrt{\Delta})$; the underlying theory of this section may be found in Section 2.2.

In order to get the appropriate floating point precision, `computeClassInvariants` first invokes algorithm `getPrecision`(Δ), which we will discuss in Chapter 9. Next, if $Q = (a, b, c)$ denotes a reduced representative of discriminant Δ, the algorithm computes $\tau_Q = (-b + i\sqrt{|\Delta|})/(2a)$. Then depending on the value of Δ mod 24 the algorithm computes one of the three following class invariants:

1. If $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$, that is if $\Delta \bmod 24 \in \{1; 17\}$, `computeClassInvariants` makes use of the modified Weber function f introduced by Yui and Zagier ([YZ97]); the definition may be found in Equation (2.21). Depending on a mod 2 and c mod 2 it computes the value $\mathfrak{f}_i(\tau_Q)$, where $\mathfrak{f}_i$ denotes the appropriate Weber function as defined in Equation (2.21); the according implementation makes use of algorithm `computeF{i}ViaEta` from the previous section. Finally, we have to consider the appropriate factor ζ_{48}^k.
2. As explained in Section 2.2, if $\Delta \not\equiv 1 \bmod 8$, $3 \nmid \Delta$, that is if $\Delta \bmod 24 \in \{4, 5, 8, 13, 16, 20\}$, `computeClassInvariants` makes use of a modified function based on γ_2 due to Atkin and Morain ([AM93]). Hence, when computing the class invariant $\zeta_3^{(a-c+a^2c)b}\gamma_2(\tau_Q)$, we make use of `computeGamma2ViaEta` from Section 7.2.5.
3. Finally, if $\Delta \equiv 0 \bmod 3$ we use `computeJViaEta` to compute the singular moduli $j(\tau_Q)$.

If Q and $g(\tau_Q)$ denote the current reduced representative and corresponding class invariant, respectively, we have $g(\tau_{-Q}) = \overline{g(\tau_Q)}$, if $-Q \neq Q$ is a reduced representative, too. Hence, we save computing about half of the class invariants. In addition, we assume that if both Q and $-Q$ belong to $C(\Delta)$, they are stored adjacently in R; we remark that the array returned by our algorithm `classGroup` has this property.

We next determine the bit-complexity of `computeClassInvariants`. As the decimal floating point precision F is fixed, we may assume that both real and imaginary part of any complex number involved is of decimal length F and thus of bitlength $\frac{\log 10}{\log 2}F = O(F)$, respectively. In order to determine the bit-complexity of `computeClassInvariants`, we have to investigate the algorithms `computeEtaViaEulerSum`, `computeF{i}ViaEta`, `computeGamma2ViaEta`, and `computeJViaEta`, respectively.

We first estimate the bit-complexity of `computeEtaViaEulerSum`(τ, F). We assume using standard algorithms for performing the underlying arithmetic. For instance, the multiplication of two complex numbers is supposed to be of complexity $O(F^2)$ and the computation of a power z^n for $z \in \mathfrak{h}$, $n \in \mathbb{N}$ is $O(\log n \cdot F^2)$ using fast exponentiation.

Lemma 7.3.1 *Let Δ be an imaginary quadratic discriminant of class number h.*
1. *Let $Q = (a, b, c)$ be a reduced representative of discriminant Δ. As usual we set $\tau_Q = \frac{-b+i\sqrt{|\Delta|}}{2a}$ and $q = e^{2\pi i \tau_Q}$. We then have $|q| \leq e^{-\sqrt{3}\pi}$.*
2. *There is an effectively computable constant $c_1 > 0$ such that $10^{-B} \leq 10^{-F}$ with $B = c_1\sqrt{|\Delta|}h$, where F denotes the precision returned by algorithm* `getPrecision`(Δ), *as explained in Chapter 9.*

Algorithm 7.11: `computeClassInvariants`(Δ, h, R)

Input: An imaginary quadratic discriminant Δ and its class number h.
An array R storing the reduced representatives of discriminant Δ.
Output: An array I holding the roots of a generating polynomial of the ring class field $L/\mathbb{Q}(\sqrt{\Delta})$.

$F \leftarrow$ `getPrecision`(Δ); $j \leftarrow 0$;
while $j < h$ **do**
  $(a, b, c) \leftarrow R[j]$; $\tau \leftarrow (-b + i\sqrt{|\Delta|})/(2a)$;
  if $\Delta \equiv 1 \bmod 8$ AND $3 \nmid \Delta$ **then**
    if $2 \mid a$ AND $2 \mid c$ **then**
      $I[j] \leftarrow \zeta_{48}^{b(a-c-ac^2)} \cdot$ `computeFViaEta`(τ, F);
    else if $2 \mid a$ AND $2 \nmid c$ **then**
      $I[j] \leftarrow (-1)^{(\Delta-1)/8} \cdot \zeta_{48}^{b(a-c-ac^2)} \cdot$ `computeF1ViaEta`(τ, F);
    else
      $I[j] \leftarrow (-1)^{(\Delta-1)/8} \cdot \zeta_{48}^{b(a-c+a^2c)} \cdot$ `computeF2ViaEta`(τ, F);
  else if $3 \nmid \Delta$ **then**
    $I[j] \leftarrow \zeta_3^{b(a-c+a^2c)} \cdot$ `computeGamma2ViaEta`(τ, F);
  else
    $I[j] \leftarrow$ `computeJViaEta`(τ, F);
  $j \leftarrow j + 1$;
  if $j < h$ AND $R[j] = -R[j-1]$ **then**
    $I[j] \leftarrow \overline{I[j-1]}$; $j \leftarrow j + 1$;
return(I);

Proof: 1. Let $(a, b, c) \in C(\Delta)$. Then, according to Proposition 6.1.1 we have $a \leq \sqrt{|\Delta|/3}$, which is equivalent to $-1/a \leq -\sqrt{3/|\Delta|}$. Hence, making use of the monotony of the e-function on the real axis we get $|q| = e^{-\pi \frac{\sqrt{|\Delta|}}{a}} \leq e^{-\sqrt{3}\pi}$.
2. Obviously we have $10^{-B} \leq 10^{-F}$ if and only if $F \leq B$. As in Chapter 9, let $L = \pi\sqrt{|\Delta|}/\log 10 \cdot \sum_{(a,b,c)\in C(\Delta)} 1/a$. The decimal precision returned by `getPrecision`(Δ) is either $F = 0.015L$ or $F = O(h/4 + L)$. We have $(1, \Delta \bmod 2, ((\Delta \bmod 2)^2 - \Delta)/4) \in C(\Delta)$ and $1 \leq a$ for all $(a, b, c) \in C(\Delta)$; hence $1 \leq \sum 1/a \leq h$ follows. Thus we deduce $F \leq c_1\sqrt{|\Delta|}h$ for some effective computable constant $c_1 > 0$. □

Algorithm `computeEtaViaEulerSum` terminates if $|q|^N < 10^{-F}$ for a positive integer N. Let N' be the positive real number with $e^{-\sqrt{3}\pi N'} = e^{-c_1\sqrt{|\Delta|}h}$. Hence, $N' = \frac{c_1}{\sqrt{3}\pi}\sqrt{|\Delta|}h = O(\sqrt{|\Delta|}h)$ follows. Lemma 7.3.1 yields $|q|^{N'} \leq e^{-\sqrt{3}\pi N'} = e^{-c_1\sqrt{|\Delta|}h} \leq 10^{-F}$. Thus, we set $N \leftarrow \lceil N' \rceil$ yielding $N = O(h^2)$. If `computeEtaViaEulerSum` terminates, we have $N \in \{\frac{3n^2+n}{2}, \frac{3n^2-n}{2}\}$ with a positive integer n. Thus, $n \sim \frac{3}{2}\sqrt{N} = O(h)$ follows.

In order to determine the bit-complexity of `computeEtaViaEulerSum` the precomputation before the while-loop is negligible. We pass at most $2n$ times through the while-loop. We have to perform at most $2n$ additions, multiplications, and computations of powers q^k with $k \leq 3n + 2$, respectively. Thus, the complexity of all multiplications is $O(n \cdot F^2) = O(h \cdot |\Delta|h^2) = O(h^5)$. In addition, the complexity of determining the q-powers is $O(n \log n \cdot F^2) = O(h \log h \cdot |\Delta|h^2) = O(h^5 \log h)$. Hence, as the addition is negligible with respect to the multiplication, the complexity of `computeEtaViaEulerSum` in terms of bit-operations is $O(h^5 \log h)$.

We next turn to the bit-complexity of algorithm `computeF2ViaEta`. With regard to algorithm `computeEtaViaEulerSum`, we have to perform in each step of the while-loop of `computeF2ViaEta` a supplementary squaring and addition, respectively. However, in all, this yields an additional addend of $n \cdot F^2$ to the complexity of `computeEtaViaEulerSum`. Thus, both complexities are equal. Furthermore, a similar argument holds for `computeFViaEta` and `computeF1ViaEta`, respectively. Next, given the value $\mathfrak{f}_2(\tau)$, we have to perform 4 squarings, 1 multiplication, 1 addition, and 1 division of complex numbers in order to compute $\gamma_2(\tau)$. Thus, the complexity of `computeGamma2ViaEta` is $O(h^5 \log h)$, too. Obviously, the same holds for `computeJViaEta`.

Finally, we determine the bit-complexity of `computeClassInvariants`. Let g denote the number-theoretical function corresponding to $\Delta \bmod 24$ as explained above. In order to get the array I, we have to evaluate at most the h values $g(\tau_Q)$. However, if Q and $-Q$ are different and both in $C(\Delta)$, we make use of $\overline{g(\tau_Q)} = g(\tau_{-Q})$. Thus we have to compute at least $h/2$ class invariants $g(\tau_Q)$. Hence, the bit-complexity to determine all roots of the class polynomial is $O(h^6 \log h)$. As we assumed to make use of standard algorithms to perform the basic computational operations we deduce the following theorem.

Theorem 7.3.2 *The bit-complexity of algorithm* `computeClassInvariants`(Δ, h, R) *is at most* $O(h^6 \log h)$.

Chapter 8

Fast Computation of Class Polynomials for Given Class Invariants

In this chapter we develop our efficient algorithm `computeClassPolynomial`. The algorithm computes a class polynomial C if the roots of C are given. More precisely, we are concerned with the following problem: Let $C \in \mathbb{Z}[X]$ be a monic and irreducible polynomial of degree h, and let $r_1, \ldots, r_h$ be its roots. Given $r_1, \ldots, r_h$, find the polynomial C. The roots r_i are returned by algorithm `computeClassInvariants` described in Chapter 7.

During the development of our efficient algorithm we compare different approaches. The result of our investigation is that although all discussed algorithms have the same asymptotic complexity, an implementation of a generalization of a multiplication method due to Karatsuba is about 25% faster in practice than the standard method.

In all we describe three approaches solving the task of this chapter; we turn to them in Section 8.2. We will show that all algorithms have an asymptotic running time of $O(h^2)$ in terms of multiplications of real numbers. Nevertheless, our most efficient algorithm `computePolynomialKaratsuba` is about 25% faster in practice than the approaches using standard techniques. Our approach bases on ideas due to Karatsuba [Kar95] and [BP01]. The most important algorithmic operation of Section 8.2 is the multiplication of monic polynomials. Thus in Section 8.1 we first develop an efficient algorithm addressing this problem. Finally, in Section 8.3 we show that the bit-complexity of algorithm `computeClassPolynomial` is $O(h^6)$.

8.1 Multiplying Polynomials

In this section we present three algorithms for multiplying monic polynomials. All approaches require two monic polynomials f and g as input and return the product $r := fg$. For the sake of generality we consider polynomials over a commutative ring R with 1. We describe our first algorithm in Section 8.1.1. This algorithm implements the standard techniques. It turns out that this approach is rather slow in practice. Second, in Section 8.1.2 we develop an algorithm which uses ideas due to Karatsuba ([Kar95]). In general this algorithm is superior

to the standard methods. However, our Karatsuba variant requires $\deg(f) = \deg(g)$. Next we compare both algorithms in Section 8.1.3. Finally, in order for the Karatsuba algorithm to be applicable for polynomials of different degree, we develop a hybrid algorithm basing on the standard and the Karatsuba algorithm in Section 8.1.4.

8.1.1 The Standard Multiplication Method

Our first algorithm to compute the product r of two monic polynomials f and g is named `productPolynomialStandard`(f, g); it bases on the standard multiplication method. We assume the polynomials to be in $R[X]$, where R denotes a commutative ring with 1. We use the notation $f = \sum_{j=0}^{k} f_j X^j$, $g = \sum_{j=0}^{l} g_j X^j$ and $r = \sum_{j=0}^{k+l} r_j X^j$, respectively. Thus we have $r_n = \sum_{j=0}^{n} f_j g_{n-j}$ for all $0 \leq n \leq k+l$, where we set $f_j = 0$ if $j > k$ and $g_j = 0$ if $j > l$, respectively. The implementation of `productPolynomialStandard` is obvious.

Algorithm 8.1: `productPolynomialStandard`(f, g)

Input: Two monic polynomials $f = \sum_{j=0}^{k} f_j X^j$ and $g = \sum_{j=0}^{l} g_j X^j$ in $R[X]$.
Output: The polynomial $r = f \cdot g$.

1: **if** $k < l$ **then**
2: $\quad t \leftarrow f$; $f \leftarrow g$; $g \leftarrow t$; //swap f and g
3: $k \leftarrow \deg(f)$; $l \leftarrow \deg(g)$; //assign the degrees of f and g to k and l, respectively
4: **for** $n = 0$ to $l - 1$ **do**
5: $\quad r_n \leftarrow \sum_{j=0}^{n} f_j \cdot g_{n-j}$;
6: **if** $k > l$ **then**
7: $\quad$ **for** $n = l$ to $k - 1$ **do**
8: $\quad\quad r_n \leftarrow f_{n-l} + \sum_{j=1}^{l} f_{n-l+j} \cdot g_{l-j}$;
9: **for** $n = k$ to $k + l - 1$ **do**
10: $\quad r_n \leftarrow f_{n-l} + \sum_{j=n-k+1}^{l-1} f_{n-j} \cdot g_j + g_{n-k}$;
11: $r \leftarrow X^{k+l} + \sum_{j=0}^{k+l-1} r_j X^j$;
12: return(r);

Obviously `productPolynomialStandard` actually returns the correct result. We next study the complexity of `productPolynomialStandard` in terms of the number of additions and multiplications in R. We prove the following result.

Proposition 8.1.1 *Let R be a commutative ring with 1. Furthermore, let f and g be two monic polynomials in $R[X]$ of degree k and l, respectively. By N_m and N_a we denote the number of multiplications and additions in R, respectively, to compute the polynomial fg using algorithm* `productPolynomialStandard`*. Then we have $N_m = N_a = kl$.*

Proof: Without loss of generality we may assume $k \geq l$. First, we count the number of multiplications $N_{m,1}$ and additions $N_{a,1}$ in the first for-loop (lines 4 - 5) of algorithm `productPolynomialStandard`. Obviously for fixed n we have to perform $n+1$ multiplications and n additions in line 5. Hence $N_{m,1} = \sum_{n=0}^{l-1}(n+1) = \frac{l(l+1)}{2}$ and $N_{a,1} = \sum_{n=0}^{l-1} n = \frac{(l-1)l}{2}$ follow.
Next we turn to the number of multiplications $N_{m,2}$ and additions $N_{a,2}$ in the second for-loop

(lines 7 - 8). For each n we have to perform l multiplications and l additions, respectively. As we have to pass $k-l$ times through the for-loop we have $N_{m,2} = N_{a,2} = (k-l)l$.
Finally, we determine the number of multiplications $N_{m,3}$ and additions $N_{a,3}$ in the third for-loop (lines 9 - 10). For fixed n we have to compute $l-(n-k+1) = k+l-n-1$ products and $l-1-(n-k+1)+2 = k+l-n$ sums in R. Thus we have

$$\begin{aligned} N_{m,3} &= \sum_{n=k}^{k+l-1} (k+l-n-1) = lk + l^2 - \sum_{n=k}^{k+l-1} n - l = \frac{(l-1)l}{2} , \\ N_{a,3} &= \sum_{n=k}^{k+l-1} (k+l-n) = N_{m,3} + l = \frac{l(l+1)}{2} . \end{aligned}$$

Thus in all we get

$$\begin{aligned} N_m &= N_{m,1} + N_{m,2} + N_{m,3} \\ &= \frac{l(l+1)}{2} + (k-l)l + \frac{(l-1)l}{2} \\ &= kl , \\ N_a &= N_{a,1} + N_{a,2} + N_{a,3} \\ &= \frac{(l-1)l}{2} + (k-l)l + \frac{l(l+1)}{2} \\ &= kl . \end{aligned}$$

This proves the proposition. □

8.1.2 A Generalization of a Multiplication Method of Karatsuba

In this section we present a generalization of an idea due to Karatsuba [Kar95]. We call this algorithm `productPolynomialKaratsuba`. Again let f and g be monic polynomials with coefficients in a commutative ring R with 1. Our algorithm `productPolynomialKaratsuba` is only applicable if the polynomials f and g are of the same degree. However, in our algorithm `computeClassPolynomial`, which is the main algorithm of this chapter, we are mostly concerned with multiplying polynomials of the same degree. Thus this is no restriction in our framework.

In the context of efficient multiplication of polynomials of degree 2 our method is already described by Bailey and Paar ([BP01]). However, we generalize the idea of Karatsuba and Bailey/Paar to polynomials of arbitrary degree.

Let k denote the degree of both f and g. We show in Proposition 8.1.2 that the number of multiplications in R to compute the product polynomial fg is equal to $k(k+1)/2$; thus compared to algorithm `productPolynomialStandard` this is only about half the number of multiplications, as we proved in Proposition 8.1.1. However, the number of additions is $k(5k-3)/2$, which is more than the corresponding number in Proposition 8.1.1. For $R = \mathbb{C}$ and $R = \mathbb{R}$ we compare the efficiency of both algorithms in practice in Section 8.1.3.

We show in Section 8.1.3 that in practice a multiplication is a lot more time-consuming than an addition. For instance, the multiplication of two real numbers is much more expensive

than an addition in $\mathbb{R}$. Depending on the floating point precision in use the factor varies from $3-40$. Thus in general the basic idea of Karatsuba is to reduce the number of multiplications in R.

For example let $k = 2$, i.e. $f = X^2 + f_1X + f_0$ and $g = X^2 + g_1X + g_0$. Using algorithm `productPolynomialStandard` Proposition 8.1.1 shows that we have to perform 4 additions and 4 multiplications in R, respectively. However, Bailey/Paar ([BP01]) show that we can proceed as follows, too: In order to determine the coefficients r_j of fg one first computes the products $d_0 = f_0g_0$ and $d_1 = f_1g_1$. Hence one has $r_0 = d_0$. In order to get $r_1 = f_0g_1 + f_1g_0$ one makes use of the relation $r_1 = (f_0 + f_1)(g_0 + g_1) - d_0 - d_1$; thus one only has to perform one multiplication in this step. However, the number of additions increases from 1 to 4. Next one determines $r_2 = d_1 + f_0 + g_0$ and $r_3 = f_1 + g_1$. Hence, to get fg one has to perform 3 multiplications and 7 additions in R. Thus, in comparison to `productPolynomialStandard`, one saves one multiplication and has to perform 3 further additions. Our algorithm `productPolynomialKaratsuba` generalizes this idea to arbitrary degree.

Algorithm 8.2: `productPolynomialKaratsuba`(f, g)

Input: Two monic polynomials $f = \sum_{j=0}^{k} f_jX^j$ and $g = \sum_{j=0}^{k} g_jX^j$ in $R[X]$.
Output: The polynomial $r = f \cdot g$.

1: $k \leftarrow \deg(f)$; //assign degree of f to k
2: **for** $n = 0$ to $k-1$ **do**
3: $\quad d_n \leftarrow f_n \cdot g_n$;
4: $r_0 \leftarrow d_0$;
5: **for** $n = 1$ to $k-1$ **do**
6: $\quad r_n \leftarrow \sum_{j=0}^{\lfloor (n-1)/2 \rfloor} (f_j + f_{n-j}) \cdot (g_j + g_{n-j}) - d_j - d_{n-j}$;
7: $\quad$ **if** $2 \mid n$ **then**
8: $\quad\quad r_n \leftarrow r_n + d_{n/2}$;
9: **for** $n = k$ to $2k-3$ **do**
10: $\quad r_n \leftarrow \sum_{j=n-k+1}^{\lfloor (n-1)/2 \rfloor} (f_j + f_{n-j}) \cdot (g_j + g_{n-j}) - d_j - d_{n-j}$;
11: $\quad$ **if** $2 \mid n$ **then**
12: $\quad\quad r_n \leftarrow r_n + d_{n/2}$;
13: $\quad r_n \leftarrow r_n + f_{n-k} + g_{n-k}$;
14: $r_{2k-2} \leftarrow f_{k-2} + d_{(2k-2)/2} + g_{k-2}$;
15: $r_{2k-1} \leftarrow f_{k-1} + g_{k-1}$;
16: $r \leftarrow X^{2k} + \sum_{j=0}^{2k-1} r_jX^j$;
17: return(r);

It is easy to see that algorithm `productPolynomialKaratsuba` is correct. We next determine the complexity of `productPolynomialKaratsuba` in terms of multiplications and additions in R, respectively.

Proposition 8.1.2 *Let R be a commutative ring with 1. Furthermore, let f and g be two monic polynomials in $R[X]$ of degree k. By N_m and N_a we denote the number of multiplications and additions in R, respectively, to compute the polynomial fg using algorithm* `productPolynomialKaratsuba`. *Then we have $N_m = \frac{k(k+1)}{2}$ and $N_a = \frac{k(5k-3)}{2}$.*

Proof: Let $N_{m,1}$ denote the number of multiplications in R in the first for-loop (lines 2 - 3). Obviously we have $N_{m,1} = k$.
Next, we turn to the for-loop in lines 5 - 8. We write $N_{m,2}$ for the number of multiplications in R and $N_{a,2}$ for the number of additions in R in this for-loop, respectively. We fix the index n in lines 6 and 8. If n is even, we have to perform $\frac{n}{2}$ multiplications and $4 \cdot \frac{n}{2} + (\frac{n}{2} - 1) + 1 = \frac{5}{2}n$ additions in R. In case of odd n we have to compute $\frac{n+1}{2}$ products and $4 \cdot \frac{n+1}{2} + (\frac{n+1}{2} - 1) = \frac{5}{2}n + \frac{3}{2}$ sums in R. Hence for even k we conclude

$$\begin{aligned} N_{m,2} &= \sum_{n=1,3,\ldots}^{k-1} \frac{n+1}{2} + \sum_{n=2,4,\ldots}^{k-2} \frac{n}{2} &&= \sum_{m=0}^{(k-2)/2} \frac{2m+2}{2} + \sum_{m=1}^{(k-2)/2} \frac{2m}{2} \\ &= \sum_{m=0}^{(k-2)/2} m + \frac{k}{2} + \sum_{m=1}^{(k-2)/2} m &&= 2 \cdot \frac{1}{2} \cdot \frac{k-2}{2} \cdot \frac{k}{2} + \frac{k}{2} \\ &= \frac{k^2}{4}\,, \end{aligned}$$

$$\begin{aligned} N_{a,2} &= \sum_{n=1,3,\ldots}^{k-1} \left(\frac{5}{2}n + \frac{3}{2}\right) + \sum_{n=2,4,\ldots}^{k-2} \frac{5}{2}n &&= \sum_{m=0}^{(k-2)/2} \left(\frac{5}{2}(2m+1) + \frac{3}{2}\right) + \sum_{m=1}^{(k-2)/2} \frac{5}{2} \cdot 2m \\ &= \sum_{m=0}^{(k-2)/2} (5m+4) + \sum_{m=1}^{(k-2)/2} 5m &&= 2 \cdot 5 \cdot \frac{1}{2} \cdot \frac{k-2}{2} \cdot \frac{k}{2} + 4 \cdot \frac{k}{2} \\ &= \frac{k(5k-2)}{4}\,. \end{aligned}$$

For odd k we have

$$\begin{aligned} N_{m,2} &= \sum_{n=1,3,\ldots}^{k-2} \frac{n+1}{2} + \sum_{n=2,4,\ldots}^{k-1} \frac{n}{2} &&= \sum_{m=0}^{(k-3)/2} \frac{2m+2}{2} + \sum_{m=1}^{(k-1)/2} m \\ &= \frac{1}{2} \cdot \frac{k-3}{2} \cdot \frac{k-1}{2} + \frac{k-1}{2} + \frac{1}{2} \cdot \frac{k-1}{2} \cdot \frac{k+1}{2} &&= \frac{k^2-1}{4}\,, \end{aligned}$$

$$\begin{aligned} N_{a,2} &= \sum_{n=1,3,\ldots}^{k-2} \left(\frac{5}{2}n + \frac{3}{2}\right) + \sum_{n=2,4,\ldots}^{k-1} \frac{5}{2}n &&= \sum_{m=0}^{(k-3)/2} (5m+4) + \sum_{m=1}^{(k-1)/2} 5m \\ &= 5 \cdot \frac{1}{2} \cdot \frac{k-3}{2} \cdot \frac{k-1}{2} + 4 \cdot \frac{k-1}{2} + 5 \cdot \frac{1}{2} \cdot \frac{k-1}{2} \cdot \frac{k+1}{2} &&= \frac{(k-1)(5k+3)}{4}\,. \end{aligned}$$

Finally, we turn to the third for-loop (lines 9 - 13). By $N_{m,3}$ and $N_{a,3}$ we denote the accumulated number of multiplications and additions in R in this loop, respectively. Again fix n. Let n be even. Then the number of multiplications in R equals $\frac{n}{2} - (n - k + 1) = k - \frac{n}{2} - 1$. Furthermore, for the number of additions we get $(4k - 2n - 4) + (k - \frac{n}{2} - 2) + 3 = 5k - \frac{5}{2}n - 3$. If n is odd the number of multiplications in R is $\frac{n+1}{2} - (n - k + 1) = k - \frac{n}{2} - \frac{1}{2}$ and the number of additions is equal to $(4k - 2n - 2) + (k - \frac{n}{2} - \frac{3}{2}) + 2 = 5k - \frac{5}{2}n - \frac{3}{2}$. Hence for even k we

have

$$\begin{aligned}
N_{m,3} &= \sum_{n=k,k+2,\ldots}^{2k-4} \left(k - \frac{n}{2} - 1\right) + \sum_{n=k+1,k+3,\ldots}^{2k-3} \left(k - \frac{n}{2} - \frac{1}{2}\right) \\
&= \sum_{m=0}^{(k-4)/2} \left(k - \frac{k+2m}{2} - 1\right) + \sum_{m=0}^{(k-4)/2} \left(k - \frac{k+2m+1}{2} - \frac{1}{2}\right) \\
&= \sum_{m=0}^{(k-4)/2} \left(\frac{k}{2} - m - 1\right) + \sum_{m=0}^{(k-4)/2} \left(\frac{k}{2} - m - 1\right) \\
&= 2 \cdot \left(\frac{k-2}{2} \cdot \frac{k}{2} - \frac{1}{2} \cdot \frac{k-4}{2} \cdot \frac{k-2}{2} - \frac{k-2}{2}\right) \\
&= \frac{k(k-2)}{4},
\end{aligned}$$

$$\begin{aligned}
N_{a,3} &= \sum_{n=k,k+2,\ldots}^{2k-4} \left(5k - \frac{5}{2}n - 3\right) + \sum_{n=k+1,k+3,\ldots}^{2k-3} \left(5k - \frac{5}{2}n - \frac{3}{2}\right) \\
&= \sum_{m=0}^{(k-4)/2} \left(5k - \frac{5}{2}(k+2m) - 3\right) + \sum_{m=0}^{(k-4)/2} \left(5k - \frac{5}{2}(k+2m+1) - \frac{3}{2}\right) \\
&= \sum_{m=0}^{(k-4)/2} \left(\frac{5}{2}k - 5m - 3\right) + \sum_{m=0}^{(k-4)/2} \left(\frac{5}{2}k - 5m - 4\right) \\
&= 2 \cdot \left(\frac{k-2}{2} \cdot \frac{5k}{2} - 5 \cdot \frac{1}{2} \cdot \frac{k-4}{2} \cdot \frac{k-2}{2} - \frac{k-2}{2} \cdot 3\right) - \frac{k-2}{2} \\
&= \frac{(k-2)(5k+6)}{4}.
\end{aligned}$$

For odd k we have

$$\begin{aligned}
N_{m,3} &= \sum_{n=k+1,k+3,\ldots}^{2k-4} \left(k - \frac{n}{2} - 1\right) + \sum_{n=k,k+2,\ldots}^{2k-3} \left(k - \frac{n}{2} - \frac{1}{2}\right) \\
&= \sum_{m=0}^{(k-5)/2} \left(k - \frac{k+1+2m}{2} - 1\right) + \sum_{m=0}^{(k-3)/2} \left(k - \frac{k+2m}{2} - \frac{1}{2}\right) \\
&= \sum_{m=0}^{(k-5)/2} \left(\frac{k}{2} - m - \frac{3}{2}\right) + \sum_{m=0}^{(k-3)/2} \left(\frac{k}{2} - m - \frac{1}{2}\right) \\
&= 2 \cdot \left(\frac{k-3}{2} \cdot \frac{k}{2} - \frac{1}{2} \cdot \frac{k-5}{2} \cdot \frac{k-3}{2} - \frac{k-3}{2} \cdot \frac{1}{2}\right) - \frac{k-3}{2} + \frac{k}{2} - \frac{k-3}{2} - \frac{1}{2} \\
&= \frac{(k-1)^2}{4},
\end{aligned}$$

$$\begin{aligned}
N_{a,3} &= \sum_{n=k+1,k+3,\ldots}^{2k-4} \left(5k - \frac{5}{2}n - 3\right) + \sum_{n=k,k+2,\ldots}^{2k-3} \left(5k - \frac{5}{2}n - \frac{3}{2}\right) \\
&= \sum_{m=0}^{(k-5)/2} \left(5k - \frac{5}{2}(k+1+2m) - 3\right) + \sum_{m=0}^{(k-3)/2} \left(5k - \frac{5}{2}(k+2m) - \frac{3}{2}\right) \\
&= \sum_{m=0}^{(k-5)/2} \left(\frac{5}{2}k - 5m - \frac{11}{2}\right) + \sum_{m=0}^{(k-3)/2} \left(\frac{5}{2}k - 5m - \frac{3}{2}\right) \\
&= \frac{k-3}{2} \cdot \frac{5}{2}k - 5 \cdot \frac{1}{2} \cdot \frac{k-5}{2} \cdot \frac{k-3}{2} - \frac{k-3}{2} \cdot \frac{11}{2} + \\
&\quad \frac{k-1}{2} \cdot \frac{5}{2}k - 5 \cdot \frac{1}{2} \cdot \frac{k-3}{2} \cdot \frac{k-1}{2} - \frac{k-1}{2} \cdot \frac{3}{2} \\
&= \frac{5k^2 - 4k - 9}{4} .
\end{aligned}$$

Obviously we have $N_m = N_{m,1} + N_{m,2} + N_{m,3}$ and $N_a = N_{a,2} + N_{a,3} + 3$. Hence for even k we get $N_m = k + \frac{k^2}{4} + \frac{k(k-2)}{4} = \frac{k(k+1)}{2}$ and $N_a = \frac{k(5k-2)}{4} + \frac{(k-2)(5k+6)}{4} + 3 = \frac{k(5k-3)}{2}$. In case of odd k we get the same results. This proves the proposition. □

8.1.3 Comparing both Algorithms

In this section we compare the performance of our algorithms `productPolynomialStandard` and `productPolynomialKaratsuba` in practice. We show that in general the latter algorithm is superior. As usual let f and g be monic polynomials in $R[X]$ of degree $k \geq 2$. In addition let C_A and C_M denote the complexity of an addition and a multiplication of two elements of R, respectively. Furthermore we write C_S and C_K for the complexity of `productPolynomialStandard` and `productPolynomialKaratsuba`, respectively. Hence, making use of Propositions 8.1.1 and 8.1.2 we get $C_S = k^2(C_A + C_M)$ and $C_K = \frac{k(5k-3)}{2}C_A + \frac{k(k+1)}{2}C_M$, respectively. We assume that the practical running time of an implementation is proportional to the complexity by some constant (positive) factor. Hence `productPolynomialKaratsuba` is faster in practice if and only if $C_K < C_S$. We have

$$\begin{array}{rcc}
C_K & < & C_S \\
\iff \quad \frac{k(5k-3)}{2}C_A + \frac{k(k+1)}{2}C_M & < & k^2(C_A + C_M) \\
\iff \quad \frac{3}{2}(k^2 - k)C_A & < & \frac{1}{2}(k^2 - k)C_M \\
\iff \quad \frac{C_A}{C_M} & < & \frac{1}{3} .
\end{array}$$

We point out that this result is independent of the degree of the polynomials. We conclude that `productPolynomialKaratsuba` is faster in practice if and only if the complexity of an addition is less than one third of the complexity of a multiplication in R.

We next compare the complexities C_A and C_M. In our application we have to deal with polynomials in $\mathbb{R}[X]$ and $\mathbb{C}[X]$. In general the coefficients are irrational numbers. Hence in general the running time does not depend on the order of magnitude of the coefficients, but on the floating point precision F we use in our computations. To determine the ratio $\frac{C_A}{C_M}$ in case of real numbers for different F we proceed as follows: We randomly choose two

numbers $r_1, r_2 \in \mathbb{R}$ represented with respect to F. We compute the sum and the product of r_1 and r_2, respectively. For each precision F we take 1000 random pairs (r_1, r_2) and sum up the CPU-time for computing the sums and products, respectively. As we assume a proportional relationship of running time and complexity with same constant for both addition and multiplication, we get the ratio $\frac{C_A}{C_M}$. The practical results can be found in Figure 8.1.

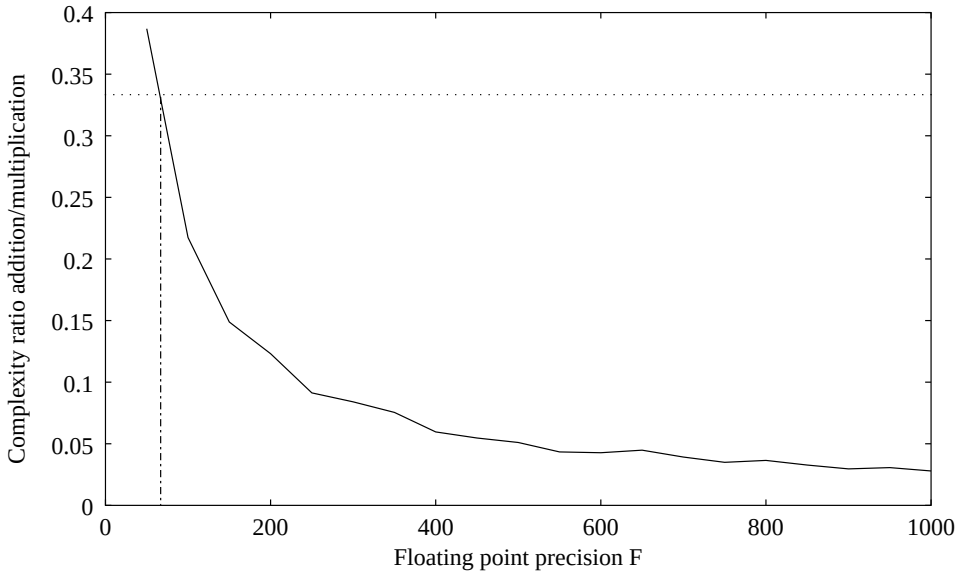


Figure 8.1: Ratio of practical running times of an addition and a multiplication of two real numbers. The ratio is plotted as a function of the floating point precision in use. The timings were measured on the SUN UltraSPARC-I.

From Figure 8.1 we deduce that $\frac{C_A}{C_M} < \frac{1}{3}$ if $F \geq 65$. Hence when multiplying two monic polynomials in $\mathbb{R}[X]$ of same degree we use algorithm `productPolynomialStandard` if and only if $F < 65$, and algorithm `productPolynomialKaratsuba` otherwise.

Finally, we consider the case of polynomials in $\mathbb{C}[X]$. A complex number is represented by its real and imaginary part. As above we denote by C_A and C_M the complexity of an addition and a multiplication of real numbers, respectively. Hence the complexity of adding two complex numbers is twice the complexity of adding to real numbers, i.e. $2C_A$. Furthermore, the complexity of multiplying two complex numbers using the standard method is equal to $2C_A+4C_M$. Thus `productPolynomialKaratsuba` is faster in practice than `productPolynomialStandard` if and only if $\frac{k(5k-3)}{2} \cdot 2C_A + (4C_M + 2C_A)\frac{k(k+1)}{2} < 2k^2C_A + (4C_M + 2C_A)k^2$, thus

$$\begin{array}{rcc} & (6k^2 - 2k)C_A + 2k(k+1)C_M & < & 4k^2(C_A + C_M) \\ \iff & 2(k^2 - k)C_A & < & 2(k^2 - k)C_M \\ \iff & \frac{C_A}{C_M} & < & 1\ . \end{array}$$

However, Figure 8.1 implies that we have $\frac{C_A}{C_M} < 1$ for all floating point precisions we make use of. Hence, in order to multiply monic complex polynomials of the same degree algorithm

`productPolynomialKaratsuba` is always superior.

8.1.4 A Hybrid Algorithm

In this section we show how to use algorithm `productPolynomialKaratsuba` if the degrees of the polynomials f and g are different. The result of this section is algorithm `productPolynomialHybrid`(f, g). It is due to G. Köhler ([Köh02]). We assume that the coefficients of f and g are elements of some commutative ring R containing 1. We explain the idea of algorithm `productPolynomialHybrid` in what follows.

As usual we denote by k and l the degrees of f and g, respectively. As the case $k = l$ is already covered by `productPolynomialKaratsuba`(f, g) we may assume $k > l$. However, in order to make use of `productPolynomialKaratsuba`(F, G) for some $F, G \in R[X]$ we have to ensure $\deg(F) = \deg(G)$. Hence we set $F = f$ and $G = g \cdot X^{k-l}$. Let P denote the polynomial returned by `productPolynomialKaratsuba`(F, G). Then we obviously have $fg = P/X^{k-l}$. From a theoretical point of view the computation of G is for free as we simply shift the coefficients of g, that is we set $G_{j+k-l} = g_j$ for $0 \leq j \leq l$ and $G_j = 0$ for $0 \leq j \leq k - l - 1$. However, we have to allocate memory for the additional coefficients and we have to perform various assignments. Furthermore, we deal with polynomials of higher degrees.

In order to decide whether we make use of `productPolynomialStandard`(f, g) or algorithm `productPolynomialKaratsuba`$(f, g \cdot X^{k-l})$ we compare the respective number of multiplications in R. In case of the standard algorithm we simply multiply f and g. Thus Proposition 8.1.1 yields that the number of multiplications in R is kl. According to Proposition 8.1.2 the corresponding number is $k(k + 1)/2$ in case of the Karatsuba method. We remark that in the latter case we take the number of multiplications of an element of R with 0 into account. The standard algorithm is superior if and only if we have $kl < k(k + 1)/2$, hence if $l \leq k/2$. However, we will make use of `productPolynomialHybrid` in the context of computing class polynomials. In order to be able to derive some closed formula for the number of multiplications and additions of our most efficient algorithm in Section 8.2.3, our algorithm `productPolynomialHybrid` invokes the standard multiplication algorithm if and only if $l < k/2$. However, the computational overhead for the case $l = k/2$ is negligible.

Algorithm 8.3: `productPolynomialHybrid`(f, g)

Input: Two monic polynomials $f = \sum_{j=0}^{k} f_j X^j$ and $g = \sum_{j=0}^{l} g_j X^j$ in $R[X]$.
Output: The polynomial $r = f \cdot g$.

if $k < l$ **then**
  $t \leftarrow f$; $f \leftarrow g$; $g \leftarrow t$; //swap f and g
$k \leftarrow \deg(f)$; $l \leftarrow \deg(g)$; //assign the degrees of f and g to k and l, respectively
if $l < k/2$ **then**
  $r \leftarrow$ `productPolynomialStandard`(f, g);
else
  $r \leftarrow$ `productPolynomialStandard`$(f, g \cdot X^{k-l})$;
  $r \leftarrow r/X^{k-l}$;
return(r);

8.2 Computing Polynomials for Given Roots

In this section we present our three algorithms to compute a polynomial if its roots are given. We denote the polynomial by R as its coefficients are in general in $\mathbb{R}$. The input of either algorithm are the roots of R, which are given as an array $(r_1, \ldots, r_h)$. For instance, if I is the array returned by algorithm `computeClassInvariants` we write $r_j = I[j-1]$ for all $1 \leq j \leq h$. Furthermore, if r_j is not real for a $j \in \{1, \ldots, h\}$, we assume that either $r_{j+1} = \overline{r_j}$ or $r_{j-1} = \overline{r_j}$.

First, in Section 8.2.1 we describe our trivial approach `computePolynomialComplex`. This algorithm simply multiplies the linear factors $X - r_j$. Next, in Section 8.2.2 we improve our first algorithm by using the relation $\overline{r_j} \in \{r_{j-1}, r_{j+1}\}$ if $r_j \notin \mathbb{R}$. Thus we only have to deal with polynomials in $\mathbb{R}[X]$. The corresponding algorithm is named `computePolynomialReal`. Next, in Section 8.2.3 we describe our algorithm `computePolynomialKaratsuba`, which bases on our algorithm `productPolynomialKaratsuba` from Section 8.1.2. Finally, in Section 8.2.4 we compare the running times of our algorithms. It turns out that `computePolynomialKaratsuba` is the most efficient approach in both theory and practice.

8.2.1 A First Approach

In this section we describe our first algorithm to compute a polynomial R if its roots are given. The algorithm is called `computePolynomialComplex`. It is rather trivial: We first initialize R with $X - r_1$. Then for each $2 \leq j \leq h$ we multiply the linear factor $X - r_j$ with the current polynomial R using algorithm `productPolynomialStandard`. Obviously this yields the correct result.

Algorithm 8.4: `computePolynomialComplex`$(r_1, \ldots, r_h)$

Input: An array $(r_1, \ldots, r_h) \in \mathbb{C}^h$. If $\mathrm{Im}(r_j) \neq 0$ we assume either $r_{j+1} = \overline{r_j}$ or $r_{j-1} = \overline{r_j}$.
Output: The monic polynomial $R \in \mathbb{R}[X]$ of degree h with $R(r_j) = 0$ for all $j \in \{1, \ldots, h\}$.

1: $R \leftarrow X - r_1$;
2: **for** $j = 2$ to h **do**
3: $\quad R \leftarrow$ `productPolynomialStandard`$(R, X - r_j)$;
4: return(R);

We remark that `computePolynomialComplex` does not make use of $\overline{r_j} \in \{r_{j-1}, r_{j+1}\}$ if $\mathrm{Im}(r_j) \neq 0$. We next analyse the complexity of `computePolynomialComplex` in terms of additions and multiplications in $\mathbb{C}$ and $\mathbb{R}$, respectively.

Proposition 8.2.1 *Let $N^{\mathbb{C}}_{a,1}$ and $N^{\mathbb{C}}_{m,1}$ denote the number of additions and multiplications of complex numbers in algorithm* `computePolynomialComplex`*, respectively. Then we have $N^{\mathbb{C}}_{a,1} = N^{\mathbb{C}}_{m,1} = \frac{1}{2}(h-1)h$. Furthermore, denote by $N_{a,1}$ and $N_{m,1}$ the number of additions and multiplications of real numbers in algorithm* `computePolynomialComplex`*, respectively. Then $N_{a,1} = N_{m,1} = 2(h-1)h$.*

Proof: I. For each $2 \leq j \leq h$ we multiply in line 3 a monic polynomial of degree $j-1$ with a monic polynomial of degree 1. Hence Proposition 8.1.1 shows that for each $2 \leq j \leq h$ we have to perform $j-1$ additions and $j-1$ multiplications in $\mathbb{C}$, respectively. Thus we have $N^{\mathbb{C}}_{a,1} = N^{\mathbb{C}}_{m,1} = \sum_{j=2}^{h}(j-1) = \frac{1}{2}(h-1)h$.
II. We represent a complex number by its real and its imaginary part. In addition, we assume to make use of standard techniques for addition and multiplication, respectively. Hence an addition of two complex numbers needs two additions of real numbers, and a multiplication of two complex numbers needs two additions and four multiplications of real numbers. Thus we conclude $N_{a,1} = N_{m,1} = 2(h-1)h$. □

Proposition 8.2.1 shows that the complexity in terms of multiplications of real numbers to compute a polynomial R, if its roots are given, is $O(h^2)$ with constant 2.

8.2.2 An Improved Approach

In this section we present our second algorithm to compute a polynomial R if its roots are given. We call the algorithm `computePolynomialReal`. We first initialize the polynomial R with the constant polynomial 1. We then improve `computePolynomialComplex` by using the property $\overline{r_j} \in \{r_{j-1}, r_{j+1}\}$, if r_j is not real. Hence let $\mathrm{Im}(r_j) \neq 0$ and $r_{j+1} = \overline{r_j}$. In line 3 of algorithm `computePolynomialComplex` we simply computed $(R \cdot (X - r_j)) \cdot (X - r_{j+1})$. Instead `computePolynomialReal` first calculates the polynomial $(X - r_j) \cdot (X - r_{j+1}) = X^2 - 2\mathrm{Re}(r_j)X + |r_j|^2$, that is the minimal polynomial of r_j over $\mathbb{R}$. Hence this polynomial has real coefficients. Then using `productPolynomialStandard` we multiply R and $X^2 - 2\mathrm{Re}(r_j)X + |r_j|^2$. Thus we only have to deal with polynomials in $\mathbb{R}[X]$. Again the correctness of `computePolynomialReal` is obvious.

Algorithm 8.5: `computePolynomialReal`$(r_1, \ldots, r_h)$

Input: An array $(r_1, \ldots, r_h) \in \mathbb{C}^h$. If $\mathrm{Im}(r_j) \neq 0$ we assume either $r_{j+1} = \overline{r_j}$ or $r_{j-1} = \overline{r_j}$.
Output: The monic polynomial $R \in \mathbb{R}[X]$ of degree h with $R(r_j) = 0$ for all $j \in \{1, \ldots, h\}$.

```
1: j ← 1;
2: R ← 1;
3: while j ≤ h do
4:    if Im(r_j) ≠ 0 then
5:       R ← productPolynomialStandard(R, X^2 − 2Re(r_j)X + |r_j|^2);
6:       j ← j + 2;
7:    else
8:       R ← productPolynomialStandard(R, X − r_j);
9:       j ← j + 1;
10: return(R);
```

As above we analyse algorithm `computePolynomialReal` in terms of additions and multiplications in $\mathbb{R}$. By n_1 we denote the number of r_j with $r_j \in \mathbb{R}$, and by n_2 the number of (up to complex conjugation different) $r_j \notin \mathbb{R}$. Hence we have $h = n_1 + 2n_2$.

Proposition 8.2.2 *Let $N_{a,2}$ and $N_{m,2}$ denote the number of additions and multiplications of real numbers in algorithm* `computePolynomialReal`*, respectively. Then we have $N_{a,2} = \frac{(h-1)h}{2}$ and $N_{m,2} = \frac{(h-1)h+2n_2}{2}$.*

Proof: I. If $\mathrm{Im}(r_j) \neq 0$ we have to perform 2 multiplications and 1 addition in $\mathbb{R}$ to compute $|r_j|^2 = \mathrm{Re}(r_j)^2 + \mathrm{Im}(r_j)^2$ in line 5. We neglect the time to compute $2\mathrm{Re}(r_j)$ as this operation is simply a shift. Hence the computation of all n_2 polynomials $X^2 - 2\mathrm{Re}(r_j)X + |r_j|^2$ takes n_2 additions and $2n_2$ multiplications in $\mathbb{R}$.
II. Let $\{j_1, j_2, \ldots, j_{n_1}\}$ be the set of all indices of real values r_j. If $j \in \{j_1, j_2, \ldots, j_{n_1}\}$ we multiply in line 8 a monic polynomial of degree $j-1$ with a monic polynomial of degree 1. Hence Proposition 8.1.1 shows that the number of additions and multiplications in $\mathbb{R}$ are both equal to $j-1$. If $j \notin \{j_1, j_2, \ldots, j_{n_1}\}$ we multiply in line 5 a monic polynomial of degree $j-1$ with a monic polynomial of degree 2. Hence again due to Proposition 8.1.1 the number of additions and multiplications in $\mathbb{R}$ are both equal to $2(j-1)$. Thus the total amount of additions and multiplications, respectively, is

$$\begin{aligned} & \sum_{j \in \{j_1, j_2, \ldots, j_{n_1}\}} (j-1) + \sum_{\substack{j \notin \{j_1, j_2, \ldots, j_{n_1}\} \\ r_{j+1} = \overline{r_j}}} 2(j-1) \\ = & \sum_{j \in \{j_1, j_2, \ldots, j_{n_1}\}} (j-1) + \sum_{\substack{j \notin \{j_1, j_2, \ldots, j_{n_1}\} \\ r_{j+1} = \overline{r_j}}} ((j-1+j)-1) \\ = & \sum_{j=1}^{h} (j-1) - n_2 = \frac{(h-1)h}{2} - n_2 \ . \end{aligned}$$

We remark that this result is independent of the position of the $r_j \in \mathbb{R}$ within the array $(r_1, \ldots, r_h)$.
III. The results of I. and II. yield $N_{a,2} = n_2 + \frac{(h-1)h}{2} - n_2 = \frac{(h-1)h}{2}$ and $N_{m,2} = 2n_2 + \frac{(h-1)h}{2} - n_2 = \frac{(h-1)h+2n_2}{2}$. □

Obviously we have $n_2 \leq \frac{h}{2}$. We claim that for odd h we have $n_1 = 1$ and $n_2 = \frac{h-1}{2} \leq \frac{h}{2}$. This is true because the elements r_j uniquely correspond to ideal classes of class groups, and we have $r_j \in \mathbb{R}$ if and only if the order of the corresponding ideal class in the class group is at most 2. However, if h is odd there is no ideal class of order 2. Hence the only ideal class yielding a real r_j is the zero element of the class group.

We conclude from Proposition 8.2.2 that $N_{m,2} \leq \frac{h^2-h+h}{2} = h^2/2$. Thus the complexity of `computePolynomialReal` in terms of the number of multiplications in $\mathbb{R}$ is $O(h^2)$, too. However, in this case the constant is $\frac{1}{2}$, and we expect algorithm `computePolynomialReal` to be four times faster than `computePolynomialComplex`. Our practical results of Section 8.2.4 confirm this assertion.

8.2.3 Computing Polynomials Using the Karatsuba Method

Finally, we turn to our third algorithm to compute a polynomial R if its roots $r_1, \ldots, r_h$ are given. The algorithm is called `computePolynomialKaratsuba`. It makes use of algorithm

`productPolynomialKaratsuba` of Section 8.1.2 and algorithm `productPolynomialHybrid` of Section 8.1.4. We show that the complexity of `computePolynomialKaratsuba` in terms of multiplications in $\mathbb{R}$ is $O(h^2)$. In Section 8.2.4 we will show that `computePolynomialKaratsuba` is faster in practice than the previous two proposed algorithms.

As we know from Section 8.1.2 `productPolynomialKaratsuba` is only applicable for polynomials of the same degree. Thus in algorithm `computePolynomialKaratsuba` we proceed as follows: As in Section 8.2.2 we write $h = n_1 + 2n_2$, where n_1 denotes the number of roots $r_j \in \mathbb{R}$. As in the proof of Proposition 8.2.2 let $\{j_1, j_2, \ldots, j_{n_1}\}$ be the set of all indices of real values r_j. Furthermore, let $\sum_{k=0}^{t} h_k 2^k$ be the binary expansion of h.

First, we compute polynomials in $\mathbb{R}[X]$ of degree 2, that is we determine $X^2 - 2\mathrm{Re}(r_j)X + |r_j|^2$ if $\mathrm{Im}(r_j) \neq 0$ and $(X - r_{j_{k_1}})(X - r_{j_{k_2}})$ if $\mathrm{Im}(r_{j_{k_1}}) = \mathrm{Im}(r_{j_{k_2}}) = 0$. As the polynomials are of degree 1 in the latter case, we make use of `productPolynomialStandard`. The polynomials are stored in an array P of length $\frac{h-h_0}{2}$.

If h is odd, we showed at the end of Section 8.2.2 that $n_1 = 1$; in this case we next initialize the result polynomial R with $X - r_{j_{n_1}}$. Next, for $1 \leq i \leq \frac{h-h_0-2h_1}{4}$ we multiply the polynomials P_{2i-1} and P_{2i} using algorithm `productPolynomialKaratsuba`. The result is a monic polynomial in $\mathbb{R}[X]$ of degree 4, respectively. The polynomials of degree 4 are stored in an array Q. If $\frac{h-h_0}{2}$ is odd, i.e. if $h_1 = 1$ there is an additional polynomial $P_{\frac{h-h_0}{2}}$ stored in P. Hence we have to multiply R and $P_{\frac{h-h_0}{2}}$. We make use of algorithm `productPolynomialHybrid` to compute their product. We then assign the array Q to the array P.

For $3 \leq j \leq t$ we proceed analogously with polynomials P_i of degree 2^{j-1}. The correctness of algorithm `computePolynomialKaratsuba` is easy to see.

We next analyse algorithm `computePolynomialKaratsuba` in terms of additions and multiplications in $\mathbb{R}$, respectively. As above we denote by n_1 the number of real r_j, and again we write $h = n_1 + 2n_2$.

Proposition 8.2.3 *Let $N_{a,3}$ and $N_{m,3}$ denote the number of additions and multiplications of real numbers in algorithm* `computePolynomialKaratsuba`*, respectively, and let $h = \sum_{k=0}^{t} h_k 2^k$ be the binary expansion of h. Then we have*

$$\begin{aligned} N_{a,3} &= n_2 + \left\lfloor \frac{n_1}{2} \right\rfloor + \frac{1}{4} \cdot \sum_{j=2}^{t} \left((5 \cdot 2^{j-1} - 3) \cdot \sum_{k=j}^{t} h_k \cdot 2^k \right) \\ &\quad + \sum_{j=2}^{t+1} h_{j-1} \cdot 2^{j-2} \cdot \left(2 \cdot (1 - h_{j-2}) \cdot \sum_{k=0}^{j-2} h_k 2^k + h_{j-2} \cdot (5 \cdot 2^{j-1} - 3) \right) . \end{aligned} \tag{8.1}$$

$$\begin{aligned} N_{m,3} &= 2n_2 + \left\lfloor \frac{n_1}{2} \right\rfloor + \frac{1}{4} \cdot \sum_{j=2}^{t} \left((2^{j-1} + 1) \cdot \sum_{k=j}^{t} h_k \cdot 2^k \right) \\ &\quad + \sum_{j=2}^{t+1} h_{j-1} \cdot 2^{j-2} \cdot \left(2 \cdot (1 - h_{j-2}) \cdot \sum_{k=0}^{j-2} h_k 2^k + h_{j-2} \cdot (2^{j-1} + 1) \right) . \end{aligned} \tag{8.2}$$

Proof: I. We first determine the number of additions and multiplications in line 7. This was already done in step I of the proof of Proposition 8.2.2. Thus the number of additions and

Algorithm 8.6: `computePolynomialKaratsuba`$(r_1, \ldots, r_h)$

Input: An array $(r_1, \ldots, r_h) \in \mathbb{C}^h$. If $\mathrm{Im}(r_j) \neq 0$ we assume either $r_{j+1} = \overline{r_j}$ or $r_{j-1} = \overline{r_j}$.
Output: The monic polynomial $R \in \mathbb{R}[X]$ of degree h with $R(r_j) = 0$ for all $j \in \{1, \ldots, h\}$.

```
 1: Compute the binary expansion sum_{k=0}^{t} h_k 2^k of h;
 2: i ← 1; j ← 1; // initialize counting variables i and j
 3: R ← 1; T ← 1; // initialize result polynomial R and temporary polynomial T
 4: P ← 0; Q ← 0; // non-initialized arrays of polynomials
 5: while j ≤ h do
 6:    if Im(r_j) ≠ 0 then
 7:       P_i ← X^2 − 2Re(r_j)X + |r_j|^2;
 8:       i ← i + 1; j ← j + 2; // increase counting variables
 9:    else
10:       if T = 1 then
11:          T ← X − r_j;
12:       else
13:          P_i ← productPolynomialStandard(T, X − r_j);
14:          T ← 1; i ← i + 1; // adapt T and increase counter i
15:       j ← j + 1;
16: if T ≠ 1 then
17:    R ← T; // h is odd
18: j ← 2; // j stores the current exponent of 2 in the binary expansion of h
19: while j ≤ t do
20:    i ← 1; // initialize i for the current loop
21:    u ← (h − sum_{k=0}^{j−1} h_k 2^k) / 2^j; // u is current upper bound of counter i
22:    while i ≤ u do
23:       Q_i ← productPolynomialKaratsuba(P_{2i−1}, P_{2i});
24:       i ← i + 1;
25:    if h_{j−1} = 1 then
26:       R ← productPolynomialHybrid(R, P_i);
27:    P ← Q; Q ← 0; // assign array of polynomial Q to P and release Q
28:    j ← j + 1;
29: R ← productPolynomialHybrid(R, P_1);
30: return(R);
```

multiplications is equal to n_2 and $2n_2$, respectively.

II. Next we turn to line 13. We multiply two monic, linear polynomials in $\mathbb{R}[X]$ using algorithm `productPolynomialStandard`. Hence according to Proposition 8.1.1 we have to perform in each pass one addition and one multiplication. Hence the total amount of both additions and multiplications is equal to $\left\lfloor \frac{n_1}{2} \right\rfloor$.

III. If h is odd we pass through line 17. However, as $R = 1$ in this case we do not have to add or multiply real numbers in this step.

IV. Now let us look at line 23. In each pass we multiply two polynomials of degree 2^{j-1} using algorithm `productPolynomialKaratsuba`. Thus, according to Proposition 8.1.2, for fixed j we have to perform $\frac{2^{j-1}(5 \cdot 2^{j-1} - 3)}{2}$ additions and $\frac{2^{j-1}(2^{j-1}+1)}{2}$ multiplications in $\mathbb{R}$, respectively. In all, for fixed j we pass $u(j) = \sum_{k=j}^{t} h_k \cdot 2^{k-j}$ times through line 23. Hence the contribution

of line 23 to the number of additions in $\mathbb{R}$ is

$$\begin{aligned}\sum_{j=2}^{t} u(j)\cdot 2^{j-2}(5\cdot 2^{j-1}-3) &= \sum_{j=2}^{t}\left(2^{j-2}(5\cdot 2^{j-1}-3)\cdot\sum_{k=j}^{t} h_k\cdot 2^{k-j}\right)\\ &= \frac{1}{4}\cdot\sum_{j=2}^{t}\left((5\cdot 2^{j-1}-3)\cdot\sum_{k=j}^{t} h_k\cdot 2^{k}\right)\ .\end{aligned}$$

Furthermore, the total amount of multiplications in $\mathbb{R}$ is

$$\sum_{j=2}^{t} u(j)\cdot 2^{j-2}(2^{j-1}+1) = \frac{1}{4}\cdot\sum_{j=2}^{t}\left((2^{j-1}+1)\cdot\sum_{k=j}^{t} h_k\cdot 2^{k}\right)\ .$$

V. We turn to the lines 26 and 29.
For each $j\in\{2,\ldots,t\}$ if $h_{j-1}\neq 0$ we multiply in line 26 monic polynomials of degree $\sum_{k=0}^{j-2} h_k 2^k$ and 2^{j-1}. Obviously we have $\sum_{k=0}^{j-2} h_k 2^k < 2^{j-1}/2$ if and only if $h_{j-2}=0$. If this is true, `productPolynomialHybrid` invokes algorithm `productPolynomialStandard`. Thus according to Proposition 8.1.1 the number of both additions and multiplications in $\mathbb{R}$ is $2^{j-1}\cdot\sum_{k=0}^{j-2} h_k 2^k$ in this case. Otherwise the number of additions and multiplications in $\mathbb{R}$ is $2^{j-2}(5\cdot 2^{j-1}-3)$ and $2^{j-2}\cdot(2^{j-1}+1)$, respectively. If $h_{j-1}\neq 0$, the number of additions for fixed j may be written as

$$\begin{aligned}2^{j-1}\cdot\sum_{k=0}^{j-2} h_k 2^k &+ h_{j-2}\cdot\left(2^{j-2}\cdot(5\cdot 2^{j-1}-3)-2^{j-1}\cdot\sum_{k=0}^{j-2} h_k 2^k\right)\\ &= 2^{j-2}\cdot\left(2\cdot(1-h_{j-2})\cdot\sum_{k=0}^{j-2} h_k 2^k + h_{j-2}\cdot(5\cdot 2^{j-1}-3)\right)\ .\end{aligned}$$

If $h_{j-1}\neq 0$, the number of multiplications for fixed j may be written as

$$\begin{aligned}2^{j-1}\cdot\sum_{k=0}^{j-2} h_k 2^k &+ h_{j-2}\cdot\left(2^{j-2}\cdot(2^{j-1}+1)-2^{j-1}\cdot\sum_{k=0}^{j-2} h_k 2^k\right)\\ &= 2^{j-2}\cdot\left(2\cdot(1-h_{j-2})\cdot\sum_{k=0}^{j-2} h_k 2^k + h_{j-2}\cdot(2^{j-1}+1)\right)\ .\end{aligned}$$

Finally, in line 29 we multiply a monic polynomial of degree $\sum_{k=0}^{t-1} h_k 2^k$ with a monic polynomial of degree 2^t. Hence we get the same formula as above for $j=t+1$. We remark that we always have $h_{j-1}=1$ in this case.
Thus the contribution of the lines 26 and 29 to the number of additions is

$$\sum_{j=2}^{t+1} h_{j-1}\cdot 2^{j-2}\cdot\left(2\cdot(1-h_{j-2})\cdot\sum_{k=0}^{j-2} h_k 2^k + h_{j-2}\cdot(5\cdot 2^{j-1}-3)\right)\ .$$

The corresponding number of multiplications is

$$\sum_{j=2}^{t+1} h_{j-1}\cdot 2^{j-2}\cdot\left(2\cdot(1-h_{j-2})\cdot\sum_{k=0}^{j-2} h_k 2^k + h_{j-2}\cdot(2^{j-1}+1)\right)\ .$$

The steps I-V yield the assertion. □

Unfortunately, Equations (8.1) and (8.2) are rather cumbersome. However, we discuss some of their properties in what follows. First, as constituted in Section 8.1.3 the running time of a multiplication is in general by far more time consuming than an addition. Thus the run time of `computePolynomialKaratsuba` is dominated by the quantity $N_{m,3}$. However, for fixed h, $N_{m,3}$ is maximal for maximal n_2. In Section 8.2.2 we explained $n_2 \leq \frac{h-2}{2}$ for even h and $n_2 = \frac{h-1}{2}$ for odd h. Hence to get upper bounds of $N_{m,3}$ in case of even h we only consider the worst case $n_2 = \frac{h-2}{2}$ in what follows.

Second, let t be fixed. In line 29 algorithm `productPolynomialStandard` is invoked if and only if $h_{t-1} = 0$. Let h_I be some class number with $2^t + 1 \leq h_I \leq 2^t + 2^{t-1} - 1$. The difference of the numbers $N_{m,3}(h_I) - N_{m,3}(h_I - 1)$ is dominated by the difference of the number of multiplications in line 29. This difference is $2^t \cdot (h_I - 2^t) - 2^t \cdot (h_I - 2^t - 1) = 2^t$. Thus $N_{m,3}(h_I) - N_{m,3}(h_I - 1)$ is approximately constant in the interval $[2^t, 2^t + 2^{t-1} - 1]$. Hence we expect $N_{m,3}$ to grow linearly in this interval. This behavior is well demonstrated in Figure 8.2. We easily recover a linear growth in the intervals $[2^{10}, 2^{10} + 2^9 - 1] = [1024, 1535]$, $[2^{11}, 2^{11} + 2^{10} - 1] = [2048, 3071]$, and $[2^{12}, 2^{12} + 2^{11} - 1] = [4096, 6143]$, respectively.

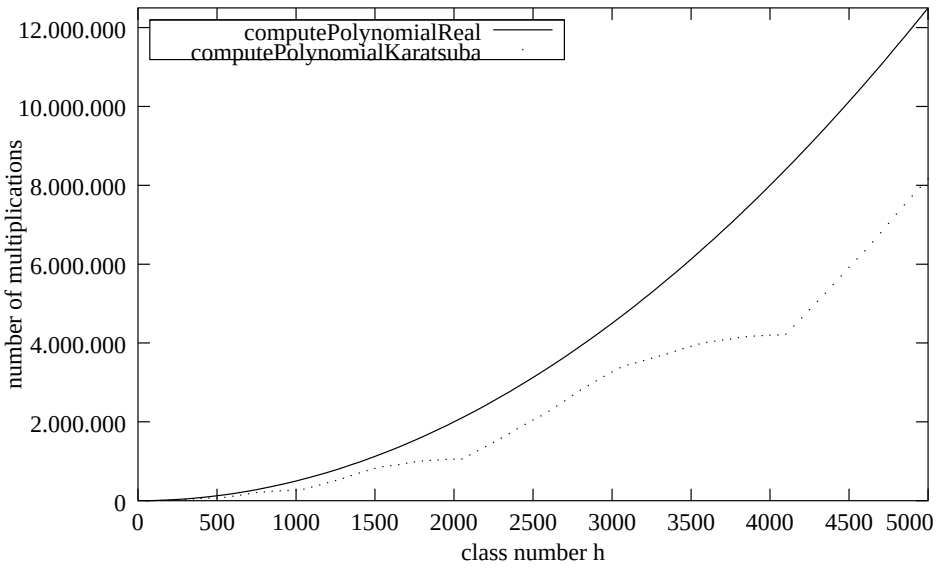


Figure 8.2: Number of multiplications of real numbers to compute a polynomial if its roots are given. We plot the corresponding numbers $N_{m,2}(h)$ and $N_{m,3}(h)$ of `computePolynomialReal` and `computePolynomialKaratsuba`, respectively, for class numbers h with $50 \leq h \leq 5000$.

Third, again we fix t. If $h_{t-1} = 1$, then `productPolynomialKaratsuba` is invoked in line 29. Let h_I be some class number with $2^t + 2^{t-1} + 1 \leq h_I \leq 2^{t+1} - 1$. The difference of the numbers $N_{m,3}(h_I) - N_{m,3}(h_I - 1)$ is now decreasing, as we deduce from Figure 8.2. We remark that the difference of the number of multiplications in line 29 of `computePolynomialKaratsuba` is

0 in this case.

Finally, we derive an upper bound of $N_{m,3}$. We first estimate the sum $\sum_{k=j}^{t} h_k \cdot 2^k$ in Equation (8.2). As we have $j \geq 2$, we conclude $\sum_{k=j}^{t} h_k \cdot 2^k \leq \sum_{k=2}^{t} 2^k = 2^{t+1} - 4$. Making use of this upper bound we get

$$\begin{aligned} \frac{1}{4} \cdot \sum_{j=2}^{t} \left((2^{j-1}+1) \cdot \sum_{k=j}^{t} h_k \cdot 2^k \right) &\leq \frac{1}{4} \cdot \sum_{j=2}^{t} \left((2^{j-1}+1) \cdot (2^{t+1}-4) \right) \\ &= (2^{t-1}-1) \cdot (2^t + t - 3) \, . \end{aligned}$$

It remains to estimate the last sum in Equation (8.2). Obviously for each j, $2 \leq j \leq t+1$, the factor in the brackets is maximal if we have $h_{j-2} = 1$. Thus the sum is bounded by $\sum_{j=2}^{t+1} 2^{j-2} \cdot (2^{j-1}+1) = 2^{2t+1}/3 + 2^t - 5/3$. Using $2^{2t+1}/3 < 2^{2t}$ we conclude $N_{m,3} = O(2^{2t})$. Hence in terms of the number of multiplications `computePolynomialKaratsuba` is of complexity $O(h^2)$, too. However, the more sparse the binary expansion of h the worse our estimation becomes.

8.2.4 Comparing the Algorithms

In this section we compare the run times of our three algorithms to compute a polynomial R if its roots are given. More precisely, for $i \in \{1,2,3\}$ let $N_{a,i}(h)$ and $N_{m,i}(h)$ denote the number of additions and multiplications in $\mathbb{R}$ of the corresponding algorithm, respectively. We present explicit values of $N_{a,i}(h)$ and $N_{m,i}(h)$ for various class numbers h, respectively. We give evidence that $N_{m,1}(h) \approx 4N_{m,2}(h)$ and $N_{m,2}(h) < N_{m,3}(h)$. In addition, we discuss the performance of either algorithm in practice. We show that `computePolynomialKaratsuba` is about 15% to 35% faster in practice than `computePolynomialReal`.

We first turn to explicit values of $N_{a,i}(h)$ and $N_{m,i}(h)$. In Table 8.1 we present data for class numbers h with $200 \leq h \leq 5000$. As usual we only consider the worst case, that is if h is even we assume $n_1 = 2$ and $n_2 = \frac{h-2}{2}$.

In addition, as mentioned above, a plot of $N_{m,2}(h)$ and $N_{m,3}(h)$ for class numbers h with $50 \leq h \leq 5000$ is given in Figure 8.2. We remark that due to Proposition 8.2.2 $N_{m,2}(h)$ grows quadratically in h. Furthermore, according to our explanation on page 130 $N_{m,3}(h)$ grows linearly in the interval $[2^t, 2^t + 2^{t-1} - 1]$.

Next, we present run times of all our three algorithms. In Table 8.2 we list CPU-timings of our approaches for class numbers up to 2500. Furthermore, we remark that the run time of `computePolynomialReal` is about a quarter of the run time of `computePolynomialComplex`. This result was already claimed at the end of Section 8.2.2. Its corresponding theoretical fraction is $\frac{N_{m,2}}{N_{m,1}} = \frac{((h-1)h+2n_2)/2}{2(h-1)h} \approx \frac{1}{4}$ by Propositions 8.2.1 and 8.2.2. In addition, we deduce from the last column in Table 8.2 that `computePolynomialKaratsuba` is about 15% to 35% faster in practice than `computePolynomialReal`.

The large speed up of about 35% comes from class numbers $2^{11} - a$ with some small $a \geq 0$. In general, the run time benefit of the Karatsuba algorithm with respect to the standard algorithm should increase in intervals $[2^t + 2^{t-1}, 2^{t+1}]$, as we deduce from our discussion at the end of Section 8.2.3 and from Figure 8.2. This behavior becomes more evident with growing floating point precision, thus with growing t. Indeed Table 8.2 supports this claim.

h	n_1	n_2	$N_{a,1}$	$N_{a,2}$	$N_{a,3}$	$N_{m,1}$	$N_{m,2}$	$N_{m,3}$
200	2	99	79600	19900	65732	79600	19999	14279
400	2	199	319200	79800	265468	319200	79999	56339
600	2	299	718800	179700	375848	718800	179999	114787
800	2	399	1278400	319600	1067552	1278400	319999	223599
1000	2	499	1998000	499500	1298436	1998000	499999	264407
1023	1	511	2091012	522753	1301759	2091012	523264	264958
1024	2	511	2095104	523776	1301760	2095104	524287	264959
1200	2	599	2877600	719400	1511660	2877600	719999	456595
1400	2	699	3917200	979300	1817016	3917200	979999	701955
1600	2	799	5116800	1279200	4282768	5116800	1279999	890479
1800	2	899	6476400	1619100	4982004	6476400	1619999	1006935
2000	2	999	7996000	1999000	5209804	7996000	1999999	1052611
2047	1	1023	8376324	2094081	5223423	8376324	2095104	1054718
2048	2	1023	8384512	2096128	5223424	8384512	2097151	1054719
2500	2	1249	12495000	3123750	6458073	12495000	3124999	2044278
5000	2	2499	49990000	12497500	25874740	49990000	12499999	8163799

Table 8.1: Number of additions and multiplications to compute a polynomial if its roots are given. As usual $N_{a,1}$ and $N_{m,1}$ denote the number of additions and multiplications in $\mathbb{R}$, respectively, using `computePolynomialComplex`. In addition, $N_{a,2}$ and $N_{m,2}$ denote the number of additions and multiplications in $\mathbb{R}$, respectively, using `computePolynomialReal`. Finally, $N_{a,3}$ and $N_{m,3}$ denote the number of additions and multiplications in $\mathbb{R}$, respectively, using `computePolynomialKaratsuba`.

8.3 The Algorithm `computeClassPolynomial`

In this section we explain our algorithm `computeClassPolynomial`(I, h). In addition, we determine its complexity in terms of bit-operations. We will show in Theorem 8.3.1 that its bit-complexity is equal to $O(h^6)$.

Input of the algorithm is an array I of length h storing the class invariants returned by algorithm `computeClassInvariants`. In addition, the algorithm requires the class number h. We stated in Section 8.2.4 that algorithm `computePolynomialKaratsuba`(I) is superior to the further algorithms of Section 8.2. Hence algorithm `computeClassPolynomial` invokes `computePolynomialKaratsuba`(I). Let R denote the polynomial returned by algorithm `computePolynomialKaratsuba`(I). As we make use of floating point arithmetic, the coefficients of R are in general not in $\mathbb{Z}$. However, if the floating point precision was chosen with care, the coefficients are very close to integers.

We introduce a method `round`(x) to get the class polynomial C from R. Let $R = \sum_{j=0}^{h} r_j X^j$, and let x denote one of the coefficients of R. Then `round`(x) returns $\lfloor x + 1/2 \rfloor$, that is the closest integer to x.

Theorem 8.3.1 *The bit-complexity of* `computeClassPolynomial`(I, h) *is at most* $O(h^6)$.

Proof: At the end of Section 8.2.3 we showed that `computePolynomialKaratsuba` requires $O(h^2)$ multiplications of real numbers. In order to estimate the bit-complexity of a multiplication we proceed as in Section 7.3. More precisely, we assume each real number to be

h	Δ	Precision	t_1	t_2	t_3	t_2/t_1	t_3/t_2
200	-21311	32	1.14	0.24	0.19	0.210526	0.791667
400	-67031	69	5.37	1.11	0.88	0.206704	0.792793
600	-148511	111	14.27	3.44	2.54	0.241065	0.738372
800	-233999	152	30.53	6.53	5.34	0.213888	0.817764
1000	-412079	203	62.84	13.58	10.95	0.216104	0.806333
1023	-363359	201	63.18	13.33	10.84	0.210984	0.813203
1024	-328319	194	63.03	13.45	10.24	0.21339	0.761338
1200	-484679	238	103.28	22.44	16.95	0.217273	0.755348
1400	-666839	287	159.91	35.96	27.79	0.224876	0.772803
1600	-970679	344	257.04	58.62	45.56	0.228058	0.777209
1800	-982511	376	367.18	84.69	64.01	0.23065	0.755815
2000	-1275599	429	515.37	120.85	81.95	0.234492	0.678113
2047	-1634279	466	603.36	142.21	90.67	0.235697	0.637578
2048	-1333631	441	566.93	133.23	86.72	0.235003	0.650904
2500	-1958519	554	1147.86	289.43	219.87	0.252147	0.759666

Table 8.2: CPU-timings of our algorithms to compute a polynomial if its roots are given. All timings are given in seconds and were measured on the Pentium III. For each h the chosen discriminant Δ is maximal with $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$. The precision comes from the formula in Table 9.1. t_1, t_2, and t_3 denote the running times of `computePolynomialComplex`, `computePolynomialReal`, and `computePolynomialKaratsuba`, respectively.

Algorithm 8.7: `computeClassPolynomial`(I, h)

Input: An array I returned by `computeClassInvariants`, and the class number h.
Output: The class polynomial $C \in \mathbb{Z}[X]$ of degree h corresponding to I.

$R \leftarrow$ `computePolynomialKaratsuba`(I);
$j \leftarrow 0$;
while $j \leq h$ **do**
  $c_j \leftarrow$ `round`(r_j); $j \leftarrow j + 1$;
return$(\sum_{j=0}^{h} c_j X^j)$;

---

of bitlength $O(F)$, where F denotes the floating point precision in use. In addition, using standard techniques for multiplying real numbers, we assume a multiplication to be of bit-complexity $O(F^2)$. Hence, making use of lemma 7.3.1, the bit-complexity of a multiplication is $O(h^4)$. As our chosen algorithm for a multiplication is assumed to be not optimal, we get an upper bound of the bit-complexity. □

Chapter 9

Precisions for Class Polynomial Computations

In this chapter we investigate the floating point precision to compute a class polynomial. We present precisions yielding a correct class polynomial H_Δ, G_Δ, and W_Δ, respectively; these polynomials are defined in Section 2.2.3. The floating point precision in use is crucial for the running time in practice. We assume that an imaginary quadratic discriminant Δ is given. The result of this chapter is our algorithm `getPrecision`(Δ) which returns the corresponding precision for either polynomial; the algorithm is explained below. As usual we write $C(\Delta)$ for the set of reduced representatives of ideal classes of discriminant Δ. An element of $C(\Delta)$ is denoted by (a, b, c).

The main result of this chapter is as follows: Let

$$L = \frac{\pi\sqrt{|\Delta|}}{\log 10} \cdot \sum_{(a,b,c)\in C(\Delta)} \frac{1}{a} \,. \tag{9.1}$$

If $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$, there is an $\alpha \in [0, 0.025]$ such that using the precision αL yields the correct Weber polynomial W_Δ. The factor α depends on the platform and the implementation of the underlying multiprecision package. Furthermore, we show that for our environment we may set $\alpha = 0.015$. This new proposed precision gives a significant speed up in practice, especially in case of discriminants having a large class number. For instance, if the class number of Δ is about 3000, our new precision has a run time benefit of about 45% with respect to previously proposed precisions, as we show in Section 9.2. In addition, we provide a proceeding to find an appropriate value of α for the current environment in use.

We restrict our thorough investigations to Weber polynomials, as only this class of polynomials is feasible for discriminants of high class numbers. Our aim is to derive an easy formula for the floating point precision to compute the Weber polynomial W_Δ. In case of the ring class polynomial H_Δ and the polynomial G_Δ we investigate whether the formulae for the precision published in articles (e.g. [LZ94]) delivers a correct polynomial. We will show that the answer actually is "yes". We next introduce two terms which we need in our further proceeding.

Definition 9.1.1 *Let C be a class polynomial. A* minimal precision *denoted by P is the minimal positive integer having the following two properties:*

1. *Using P as a floating point precision in* `cryptoCurve` *yields the correct polynomial C.*
2. *For all precisions F with $F \geq P$ we get the correct polynomial, too.*

A sufficient precision *is a floating point precision yielding the correct polynomial C.*

Obviously we have $P = \max\{F \in \mathbb{N} : F \text{ is not a sufficient precision}\}+1$. Unfortunately, there is no theoretically proven formula for a sufficient precision in case of any class polynomial. Hence, we make use of some heuristics to justify our formula αL as a sufficient precision to compute W_Δ. Nevertheless, as shown in Section 9.2 our result bases on the computation of P for a large number of discriminants. Our proposals for the precisions to compute a class polynomial may be found in Table 9.1. The precisions in case of G_Δ and H_Δ are due to Lay/Zimmer ([LZ94]), respectively.

Condition on Δ	Class Polynomial	proposed precision
$\Delta \equiv 1 \bmod 8$, $\Delta \not\equiv 0 \bmod 3$	W_Δ	$\lceil 0.015L \rceil$
$\Delta \not\equiv 0 \bmod 3$	G_Δ	$\lceil (L + h/4 + 5)/3 \rceil$
None	H_Δ	$L + \lceil h/4 \rceil + 5$

Table 9.1: Sufficient precisions to compute a class polynomial W_Δ, G_Δ, and H_Δ, respectively.

Our algorithm `getPrecision`(Δ) is quite easy to explain. It requires an imaginary quadratic discriminant and, depending on the value of $\Delta \bmod 24$, returns a sufficient precision to compute the class polynomial corresponding to Δ as listed in Table 9.1.

This chapter is organized as follows: As our main formula αL bases on observations on the ring class polynomial H_Δ, we first review the precision to compute H_Δ in Section 9.1. In addition, Section 9.1 deals with the precision to determine G_Δ. Next, in Section 9.2 we investigate in detail the precision to compute Weber polynomials.

9.1 Sufficient Precisions to Compute the Class Polynomials H_Δ and G_Δ

In this section we derive a formula for a sufficient precision to compute the ring class polynomial H_Δ. We discuss two proposals, that is the precision proposed by [AM93]/[BSS99] and the precision which may be found in [LZ94]. However, none of them gives evidence that their proposal actually is appropriate. We present practical data supporting the formula of Lay/Zimmer [LZ94]. In addition, we present a formula for the precision in case of the class polynomial G_Δ. As usual we write H and G instead of H_Δ and G_Δ, respectively.

Similar to [AM93]/[BSS99] we start with some heuristics to get a proposal for the floating point precision of H. We first recall that for a reduced representative $Q = (a, b, c)$ we set $\tau_Q = (-b + i\sqrt{|\Delta|})/(2a)$ and $q = e^{2\pi i \tau_Q}$. Thus $|q| = e^{-\pi\sqrt{|\Delta|}/a}$. Hence, for fixed Δ, $|q|$ only depends on a, and using the Fourier series of j (see Section 7.1.1) we get $|j(\tau_Q)| \approx |1/q| = e^{\pi\sqrt{|\Delta|}/a}$. We deduce that the constant term of H is up to sign of order of magnitude $e^{\pi\sqrt{|\Delta|}\sum_{(a,b,c)\in C(\Delta)} 1/a}$. In most cases the constant term of H is up to sign the biggest coefficient; for instance, we refer to the third column of Table 9.2. Furthermore, its decimal length can be approximated

by Equation (9.1). In order to compute H two different proposals for sufficient precisions F are known:

1. Atkin, Morain ([AM93]) and Blake, Seroussi, Smart ([BSS99]): $F = \binom{h}{\lfloor h/2 \rfloor} \cdot L + 10$,
2. Lay, Zimmer ([LZ94]): $F = L + h/4 + 5$.

The sufficient precision proposed in [AM93] and [BSS99] bases on the observation that the coefficients of H are symmetric functions in the h singular moduli $j(\tau_Q)$. More precisely, if we write the ring class polynomial H as $H = \sum_{i=0}^{h} a_i X^i$, we have $a_h = 1$ and

$$a_{h-i} = (-1)^i \cdot \sum_{1 \le k_1 < \ldots < k_i \le h} j(\tau_{Q_{k_1}}) \cdot \ldots \cdot j(\tau_{Q_{k_i}}) \,, \qquad 1 \le i \le h \,. \tag{9.2}$$

Let i be fixed. Then the decimal length of the absolute value of each addend in (9.2) is bounded by L. Furthermore, there are at most $\binom{h}{\lfloor h/2 \rfloor}$ addends, namely for $i = \lfloor h/2 \rfloor$. Thus $\binom{h}{\lfloor h/2 \rfloor} \cdot L$ is an upper bound of the decimal length of $|a_i|$. However, it turns out that this upper bound is not applicable in practice and rather bad; for example, if $\Delta = -21311$ we compute

$$\binom{h}{\lfloor h/2 \rfloor} \cdot L + 10 = 9.05485 \cdot 10^{58} \cdot 2107.02 + 10 = 1.90876 \cdot 10^{62} \,.$$

However, Lay and Zimmer only estimate the decimal length of $|a_0|$; in addition, they vary it by adding a linear term in h. Obviously, their proposed sufficient precision is much lower than the value proposed by [AM93] and [BSS99]. Again, looking at $\Delta = -21311$, their proposal gives a sufficient precision of 2163 .

Basing on the sufficient precision of Lay and Zimmer we computed some dozens of ring class polynomials using the precision $F = L + \lceil h/4 \rceil + 5$. The discriminants are divisible by 3 and have even class numbers in the interval $[200, 208]$. More precisely, for each even class number $h \in [200, 208]$ we computed H for the n largest discriminants of class number h, where $18 \le n \le 34$. In any case we actually get the ring class polynomial; we decide whether H is the ring class polynomial by using a probabilistic correctness test similar to the test `isWeberPolynomial` which we will present in Section 9.2. Furthermore, we always have $F > \max\{\log_{10} |a_i| : 0 \le i \le h\}$. Our results are listed in Table 9.2. We conclude that in case of H the precision of Table 9.1 is sufficient.

Finally, we turn to the computation of a polynomial G. The roots of G are of the form $\zeta_3^k \gamma_2(\tau_Q)$ where k is an integer. The relation $j(\tau) = \gamma_2(\tau)^3$ shows $|j(\tau)| = |\gamma_2(\tau)|^3$. Thus, if F is a sufficient precision to compute H, the precision $F/3$ should be a sufficient precision to determine G. As in the case of the ring class polynomial we determined dozens of polynomials G using the precision of Table 9.1, and in either case we get the correct polynomial. We remark that this precision was already proposed by Lay and Zimmer ([LZ94]).

9.2 Precision to Compute Weber Polynomials

In this section we investigate sufficient precisions to compute the Weber polynomial W_Δ in practice. We simply write W for a Weber polynomial in what follows. Our result is that for a given discriminant Δ with $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$, αL is a sufficient precision, where L is defined in Equation (9.1) and $\alpha \in [0, 0.025]$. This precision is much lower than the precision proposed by Lay and Zimmer ([LZ94]). We show that an implementation using our new precision is

h	number of tested Δ	number of Δ, for which $\lvert a_0\rvert$ is not maximal	sample Δ	$\max\lfloor\log_{10}\lvert a_i\rvert\rfloor+1$, $0\le i\le h$	index i of $\max\lvert a_i\rvert$	F
200	18	1	-29559	2187	0	2245
			-48351	2485	2	2547
202	34	3	-35439	2347	0	2409
			-341244	3122	2	3190
204	28	3	-31191	2260	0	2319
			-36039	2343	1	2406
206	18	3	-51351	2697	0	2756
			-113991	3663	4	3731
208	18	1	-34359	2341	0	2403
			-43911	2397	1	2460

Table 9.2: Statistics for the coefficients of some ring class polynomials.

about 45% faster than the same implementation using the precision of Lay and Zimmer. The factor α depends on the environment in use, that is the multiprecision package, the computer algebra library, and the compiler. At the end of this section we describe a proceeding to find an appropriate value α for the current environment.

As in case of the ring class polynomial H we present some heuristics for deriving our main formula αL. The roots of W are of the form $\zeta_{48}^k g(\tau_Q)$ where k is an integer and $g\in\{\mathfrak{f},\mathfrak{f}_1,\mathfrak{f}_2\}$. We have

$$j(\tau)=\frac{(\mathfrak{f}(\tau)^{24}-16)^3}{\mathfrak{f}(\tau)^{24}}=\frac{(\mathfrak{f}_1(\tau)^{24}+16)^3}{\mathfrak{f}_1(\tau)^{24}}=\frac{(\mathfrak{f}_2(\tau)^{24}+16)^3}{\mathfrak{f}_2(\tau)^{24}}\ .$$

Hence, assuming $|g(\tau)|\gg 1$, we estimate $|j(\tau)|\approx|g(\tau)|^{48}$. Thus, if F is a sufficient precision to compute the ring class polynomial, $F/48$ should be a sufficient precision to determine W. Indeed, Lay and Zimmer claim that $(L+h/4+5)/47+1$ is a sufficient precision for Weber polynomials.

However, we present data showing that even αL is a sufficient precision. As already mentioned the factor α depends on the environment in use. As shown below, for our environment we may set $\alpha=0.0150<\frac{1}{48}=0.0208$. Using this precision we get a significant speed up in practice to compute W. In order to confirm our claim, we identified the minimal precision P for a large set of discriminants.

We explain how we determine the minimal precision P for a given discriminant Δ. This task is accomplished by our algorithm `findMinimalPrecision`(Δ). Its input is an imaginary quadratic discriminant Δ with $\Delta\equiv 1 \bmod 8$ and $\Delta\not\equiv 0 \bmod 3$. The algorithm outputs the minimal precision P for the used environment to compute the class polynomial W.

We first describe how to get the Weber polynomial W. We are convinced that W actually is the Weber polynomial if our probabilistic correctness test `isWeberPolynomial`(Δ, W) returns `true`. The algorithm requires an imaginary quadratic discriminant Δ with $\Delta\equiv 1 \bmod 8$, $3\nmid\Delta$. In addition, it gets a monic polynomial $W\in\mathbb{Z}[X]$ as input. `isWeberPolynomial` proceeds as follows: It first determines a prime p of bitlength about 60 which is suitable for the CM-method of discriminant Δ. As usual we write $p=(t^2-\Delta y^2)/4$ with $(t,y)\in\mathbb{N}^2$. If $W \bmod p$ does not have a root in $\mathbb{F}_p$, the algorithm returns `false`. Otherwise, let c_p denote such a root. Making use of c_p the algorithm computes twisted elliptic curves E_1 and E_2 over

$\mathbb{F}_p$ as explained at the end of Section 2.3.4. If W is the Weber polynomial, one of the curves is of order $p+1-t$. Let $P_1 \in E_1(\mathbb{F}_p) \setminus \{O\}$ and $P_2 \in E_2(\mathbb{F}_p) \setminus \{O\}$ be randomly chosen points on E_1 and E_2 over $\mathbb{F}_p$, respectively. If $[p+1-t]P_1 \neq O$ and $[p+1-t]P_2 \neq O$, `isWeberPolynomial` returns `false`. Otherwise it returns `true`. The probability of accepting a false polynomial is negligible in practice.

In order to compute W we make use of our algorithm `computePolynomial`(Δ, F). This algorithm mainly consists of the algorithms `classGroup`(Δ), `computeClassInvariants`(Δ, h, R), and `computeClassPolynomial`(I, h). However, in `computeClassInvariants`(Δ, h, R) we set the floating point precision to F instead of invoking `getPrecision`(Δ). Our initial choice of F is the sufficient precision proposed by Lay and Zimmer. We are not aware of any example where the computation within this precision failed. However, if `isWeberPolynomial` returns `false` for the first computed polynomial we iteratively increase F by 1 until we find a polynomial passing `isWeberPolynomial`. In this case, `findMinimalPrecision` returns the last used precision. If our first computed polynomial turns out to be the Weber polynomial, we successively decrease the precision by 1 and compute a polynomial $\hat{W}$ within this precision, until we have $W \neq \hat{W}$ for the first time. In this case, `findMinimalPrecision` returns the last but one precision.

Algorithm 9.1: `findMinimalPrecision`(Δ)

Input: An imaginary quadratic discriminant Δ with $\Delta \equiv 1 \bmod 8$ and $\Delta \not\equiv 0 \bmod 3$.
Output: The minimal precision P yielding the Weber polynomial W corresponding to Δ.

$L \leftarrow \frac{\pi\sqrt{|\Delta|}}{\log 10} \cdot \sum_{(a,b,c)\in C(\Delta)} \frac{1}{a}$; $F \leftarrow \lceil (L + h/4 + 5)/47 + 1 \rceil$;
$W \leftarrow$ `computePolynomial`(Δ, F);
if `isWeberPolynomial`$(\Delta, W) =$ `false` **then**
  while `true` **do**
    $F \leftarrow F + 1$;
    $W \leftarrow$ `computePolynomial`(Δ, F);
    if `isWeberPolynomial`$(\Delta, W) =$ `true` **then**
      return (F);
else
  while `true` **do**
    $F \leftarrow F - 1$;
    $\hat{W} \leftarrow$ `computePolynomial`(Δ, F);
    if $W \neq \hat{W}$ **then**
      return $(F + 1)$;

We derive our main formula $F = \alpha L$. Using algorithm `findMinimalPrecision`(Δ) we computed the minimal precision P for lots of discriminants Δ as listed in Table 9.3. We remark that we have $\Delta \equiv 1 \bmod 8$ and $\Delta \not\equiv 0 \bmod 3$ for all discriminants.

In all we tested 62103 discriminants of various different class numbers. All tests were performed on the SUN UltraSPARC-IIi. In Figure 9.1 we plot P/L as a function of L for all these discriminants. We see that we have $P/L < 0.015$ for all tested discriminants. Hence, $\alpha = 0.015$ seems to be a good choice. In addition, for some randomly chosen discriminants we determined P on the Pentium III, too; we got the same results as on the SUN UltraSPARC-IIi. Furthermore, we successfully used our precision αL to compute the Weber polynomials

h	condition on Δ	number of tested Δ
200 to 499	$\lvert\Delta\rvert \leq 6 \cdot 10^6$	51603
500 to 999	the first 20 discriminants	10000
1000 to 1495, $5 \mid h$	the first 5 discriminants	500

Table 9.3: Test discriminants to derive the precision for Weber polynomials. All discriminants are congruent 1 modulo 8 and not divisible by 3.

for discriminants of large class number in Section 11.1.2, that is for discriminants of class numbers up to 15000.

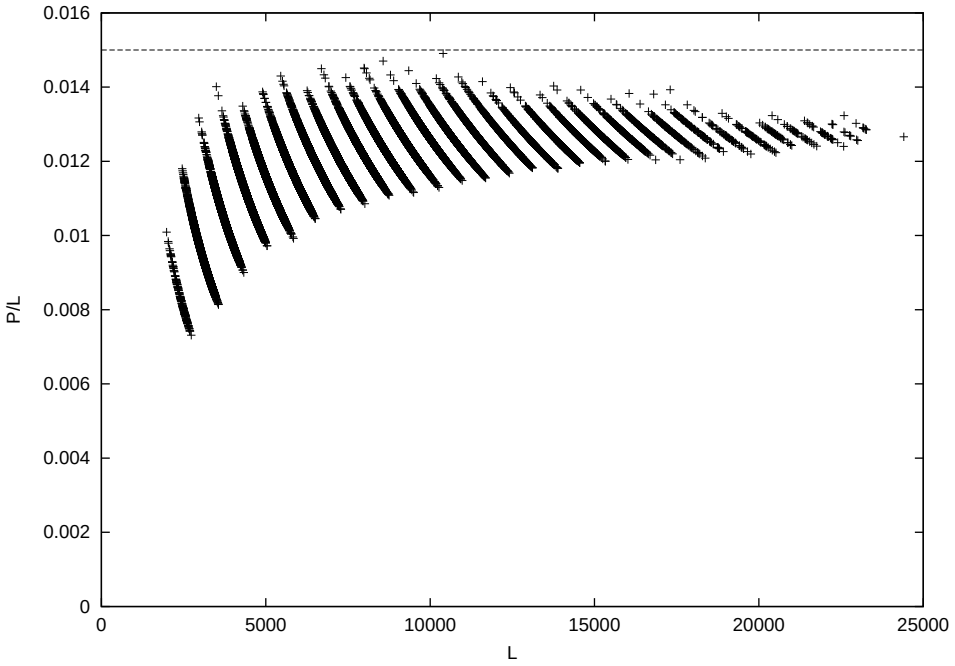


Figure 9.1: Minimal precisions P to compute a class polynomial W. We plot P/L as a function of L for all discriminants of Table 9.3.

We next demonstrate the run time benefit of our new floating point precision to compute W. Let F_L denote the precision proposed by Lay/Zimmer, that is we have $F_L = \lceil (L + h/4 + 5)/47 + 1 \rceil$. We choose discriminants of class numbers in the range 3000 to 5000. More precisely, using our database delta_h which stores discriminants according to their class numbers, for each class number h with $3000 \leq h \leq 5000$, $500 \mid h$ we determined the maximal discriminant Δ of class number h with the boundary conditions $\Delta \equiv 1 \bmod 8$ and $3 \nmid \Delta$. If Δ denotes such a discriminant, we invoke `computePolynomial`(Δ, F) using $F = \lceil \alpha L \rceil$ and $F = F_L$, respectively. All our timings are measured on the Pentium III. The results are given in Table 9.4.

We deduce from Table 9.4 that the fraction of our new precision to the precision F_L of Lay/Zimmer is approximately constant in the tested interval. In addition, we see that our

h	Δ	$\lceil \alpha L \rceil$	t_α	F_L	t_L	$\lceil \alpha L \rceil / F_L$	t_α / t_L
3000	-2668511	678	515.98	978	939.89	0.69325	0.54898
3500	-3670631	811	910.49	1171	1725.9	0.69257	0.52755
4000	-4728671	941	1659.8	1357	2516.4	0.69344	0.65959
4500	-5899871	1076	1973.3	1552	3355.3	0.69330	0.58811
5000	-6961631	1195	2703.4	1722	5116.6	0.69396	0.52836

Table 9.4: Run time benefit of our new proposed precision αL to compute a class polynomial W. All discriminants are maximal of the given class number with $\Delta \equiv 1 \bmod 8$ and $3 \nmid \Delta$. t_α and t_L denote the timing of `computePolynomial`(Δ, F) with $F = \lceil \alpha L \rceil$ and $F = F_L$ in seconds on the Pentium III, respectively.

precision yields a run time benefit of up to 47% for discriminants of large class numbers. This is a rather promising result to compute Weber class polynomials corresponding to discriminants of large class numbers.

Finally, we define a proceeding to determine an appropriate value of α for the current environment in use. In our above sample of discriminants from Table 9.3 we find the following discriminants of largest values P/L; these values of P/L may be seen in Figure 9.1.

Δ	P/L	h	Δ	P/L	h
-582223	0.014299	253	-759247	0.0144964	316
-1648903	0.0143295	400	-412367	0.0144966	436
-1133647	0.0143332	312	-1026823	0.0145172	384
-1109263	0.0143841	390	-1387687	0.0147036	384
-1706023	0.0144432	408	-2334655	0.0149082	424

Table 9.5: Largest values of P/L for the discriminants of Table 9.3.

We assume that the largest values P/L are the same for all environments. Hence we first determine the fraction P/L for all discriminants of Table 9.5. The maximum of these values is assumed to be in $[0, 0.025]$. Then we simply choose

$$\alpha = \frac{\lceil 1000 \cdot \max\{P/L\} \rceil}{1000} .$$

This gives an appropriate choice of α. The program find_appropriate_alpha in the appl-directory of our library gec implements this proceeding and may be used in order to determine an appropriate value α for the environment in use.

Chapter 10

Complexity and Security Considerations

We investigate the bit-complexity of our main algorithm `cryptoCurve`(r_0, k_0, h_0). We develop a closed formula of the bit-complexity of `cryptoCurve` in Section 10.1. The result is Formula (10.1). Next, in Section 10.2 we discuss the number of "different" elliptic curves which may be output by `cryptoCurve`(r_0, k_0, h_0) if the input is fixed. The term "different" depends on the underlying attacker model. We discuss two rather strong attacker models and conclude that in each model the number of different curves is large. This has an important implication with respect to security as the curves returned by `cryptoCurve` are not special and hence seem to be not amenable to special attacks.

10.1 Complexity of `cryptoCurve`

We derive the bit-complexity of our main algorithm `cryptoCurve`(r_0, k_0, h_0). As explained in Chapter 3, our main algorithm consists of the subalgorithms `findPrime`(r_0, k_0, h_0) and `findCurve`(Δ, p, r, k), respectively. Hence, we have to determine the respective bit-complexity.

Let us first consider algorithm `findPrime`. Depending on whether a prime field is chosen, it either invokes `findPrimeDeltaFixed` or `findDiscriminant`. However, as explained in Chapter 5 we were not able to determine the complexity of the latter algorithm. Thus our following discussion is only valid if `findPrime` invokes `findPrimeDeltaFixed`. In addition, we assume that Δ is chosen such that $h(\Delta) = h_0$. Using the reasonable assumptions of Gross and Smith in [GS00] we proved in Theorem 4.3.3 that if $\Delta \equiv 5 \bmod 8$ the bit-complexity of finding a generalized twin prime pair $(\pi, \pi + 1) \in \mathcal{O}_\Delta$ with $\log_2 \pi\overline{\pi} = \log_2 r_0 k_0$ is $O(h_0^2 \cdot \log^5 r_0 k_0 / |\Delta|)$. Furthermore, we assumed that this complexity is an upper bound of the complexity of all optimized algorithms in Chapter 4.3.1, 4.3.2, and 4.3.3, respectively. We justified these assumptions in the corresponding sections. Hence making use of Theorem 2.1.17 we conclude $O(\log^5 r_0 k_0)$ for the bit-complexity of `findPrime`.

We next turn to algorithm `findCurve`. The most time-consuming step of this subalgorithm is algorithm `findRoot`. Besides algorithm `find_root` the bit-complexity of either subalgorithm of `findRoot` is already determined. In Theorem 10.1.1 we present the bit-complexity of `find_root`.

Theorem 10.1.1 *Let p be a rational prime and $C \in \mathbb{Z}[X]$ a polynomial of degree d which splits modulo p into d different linear polynomials. The bit-complexity of* `find_root`(p, C) *is* $O(d^2 \log d \log\log d \cdot \log^4 p)$.

Proof: As stated in Chapter 3 algorithm `find_root` makes use of a very efficient polynomial arithmetic due to Shoup. In his paper [Sho95] Shoup investigates the complexity of his algorithm in terms of scalar operations modulo p. On page 4 in [Sho95] he states that his algorithm takes $O(d^2 \log d + d^2 \log d \log\log d \cdot \log p)$ scalar operations in $\mathbb{F}_p$. However, such a scalar operation mainly consists of a Euclidian step and is thus of bit-complexity at most $O(\log^3 p)$ (see [Coh95], p.13). Asymptotically this proves the assertion. □

In Table 10.1 we summarize the bit-complexities of the subalgorithms of `findRoot`.

Sub-algorithm	Bit-complexity	Reference
`classGroup`	$O(h_0^2 \log^3 h_0)$	Proposition 6.1.2
`computeClassInvariants`	$O(h_0^6 \log h_0)$	Theorem 7.3.2
`computeClassPolynomial`	$O(h_0^6)$	Theorem 8.3.1
`find_root`	$O(h_0^2 \log h_0 \log\log h_0 \cdot \log^4 r_0 k_0)$	Theorem 10.1.1

Table 10.1: Bit-complexities of the subalgorithms of `findRoot`

The final computation of a root j_p of the ring class polynomial H modulo p in `findRoot` is of bit-complexity $\log^3 r_0 k_0$ and thus can be neglected. The same holds for the decision in algorithm `findCurve` which of the twisted elliptic curves is of order rk. Thus, in all we have shown the following theorem.

Theorem 10.1.2 *If no prime field is chosen in algorithm* `findPrime`*, the bit-complexity of algorithm* `cryptoCurve`(r_0, k_0, h_0) *is at most*

$$O(\log^4 r_0 k_0 (\log r_0 k_0 + h_0^2 \log h_0 \log\log h_0) + h_0^6 \log h_0) \; . \tag{10.1}$$

10.2 Security Considerations

In this section we investigate the number of different elliptic curves that may be returned by `cryptoCurve`(r_0, k_0, h_0) for some fixed input (r_0, k_0, h_0). We call such a curve a *reachable elliptic curve* for (r_0, k_0, h_0). More precisely, let (r_0, k_0, h_0) be given. Our algorithm outputs a curve having a maximal order of class number at least h_0 as endomorphism ring. If no prime field is set in algorithm `findPrime`, there are no further restrictions for the discriminant of the endomorphism ring. However, if a field is chosen, we have to consider the boundary conditions $\Delta > -6.000.000$ and $h(\Delta) \leq 2000$. In addition, for efficiency reasons we assume $3 \nmid \Delta$, that is the Hilbert class field is generated by the polynomial G. We call an elliptic curve *reachable* if it respects the requirements for Δ.

Conversely, if such a curve is given, it is easy to see that this curve is the output of our algorithm with a non-vanishing probability. Let E denote this curve and let $\mathbb{F}_p$ be its field of definition. We furthermore assume, that its cardinality N and factorization $N = rk$

are known. Otherwise, we make use of the efficient SEA-algorithm to determine N; using trial division we find k and finally r. Next, using $(p+1-N)^2-4p$ we easily compute the fundamental discriminant Δ. This is true since $|\Delta|$ is the minimal divisor of $|(p+1-N)^2-4p|$ such that $-|\Delta|$ is an imaginary quadratic discriminant. Once knowing Δ, p, r, and k we invoke `findCurve`(Δ, p, r, k). Let j_p denote the root of H mod p returned by algorithm `findRoot`, and finally let E' denote the elliptic curve returned by `cryptoCurve`. With probability $1/h$ the j-invariants of E and E' are the same; hence assuming that a $\mathbb{F}_p$-twisted curve of E is not cryptographically strong, with probability $1/h$ E and E' are $\mathbb{F}_q$-isomorph and may thus be regarded as the same curve.

An important implication with respect to security is that the more curves are reachable the less "special" and hence more secure the curves are. We distinguish two forms of attacks. For each model of attacks we count the number of different reachable curves. We get different results in either model. However, to our knowledge both models are rather strong, that is no real attacker is able to perform the described attacks in practice. Hence, in practice the number of different curves is larger than in either attacker model.

We investigate the following attacker models: Our first model of attacks assumes that the adversary is able to compute the lift $\mathcal{E}$ of E over the Hilbert class field and solves the ECDLP in $\mathcal{E}$. We call this scenario the lift-attack model, and we discuss the number of different curves in this model in Section 10.2.1. We remark that the lift-attack model is the strictest security model and that there is no evidence that a lift-attack will ever be feasible. Next, we assume that the ECDLP only depends on the j-invariant of the curve. More precisely, given two elliptic curves E_1 and E_2 over $\mathbb{F}_p$ with $j(E_1) = j(E_2)$, the ECDLP in $E_1(\mathbb{F}_p)$ may be reduced to the ECDLP in $E_2(\mathbb{F}_p)$ in polynomial time. This model is called the j-attack model. We discuss the j-attack model in Section 10.2.2.

In both cases we show that there is a large number of different elliptic curves which may be returned by our algorithm. Throughout this section let G be a point of order r in $E(\mathbb{F}_p)$ and for some l with $1 \leq l \leq r$ we set $P = [l]G$. Hence, given G and P the ECDLP consists in computing the integer l.

10.2.1 Number of Curves in the Lift-Attack Model

In this section we discuss the number of different elliptic curves over a prime field in the lift-attack model. We first explain the lift of an elliptic curve. Let E be an elliptic curve over a prime field $\mathbb{F}_p$ with $\mathrm{End}(E) = \mathcal{O}_\Delta$, where $\mathcal{O}_\Delta$ is a maximal order. In addition, let L be the Hilbert class field of $\mathcal{O}_\Delta$. A *lift of E over L* is an elliptic curve $\mathcal{E}$ defined over L with $\mathrm{End}(\mathcal{E}) = \mathcal{O}_\Delta$ and such that there is a prime $\mathfrak{P}$ of L over $p\mathbb{Z}$ with $\mathcal{O}_L/\mathfrak{P} = \mathbb{F}_p$ and $E = \mathcal{E} \bmod \mathfrak{P}$.

In the lift-attack model we assume that an adversary is able to perform some attack on a lift $\mathcal{E}$ of E. More precisely, the attacker computes lifted points $\mathcal{G}$ of G and $\mathcal{H}$ of H in $\mathcal{E}(L)$, respectively, and solves the ECDLP for $\mathcal{G}$ and $\mathcal{H}$. We have the following commutative diagram.

$$\begin{array}{ccc} \mathcal{G} & \xrightarrow{[l]\mathcal{G}} & \mathcal{P} \\ \uparrow \text{lift} & & \uparrow \text{lift} \\ G & \xrightarrow{[l]G} & P \end{array}$$

Hence, two curves are different in the lift-attack model if and only if their endomorphism rings have different discriminants. Thus we have to count all fundamental discriminants Δ of class number between 200 and 2000 such that $3 \nmid \Delta$ and $\Delta > -6.000.000$. As we have different lower bounds of the cofactors for $\Delta \equiv 0 \bmod 4$, $\Delta \equiv 1 \bmod 8$, and $\Delta \equiv 5 \bmod 8$, respectively, we distinguish these three cases. Let $D_0(h)$, $D_1(h)$, and $D_5(h)$, denote the number of fundamental discriminants $\Delta \equiv 0 \bmod 4$, $\Delta \equiv 1 \bmod 8$, and $\Delta \equiv 5 \bmod 8$ of class number h, respectively, having the properties cited above. For example, we have $D_0(200) = 1484$, $D_1(200) = 200$, and $D_5(200) = 1772$. The numbers $D_i(h)$ are rather volatile. Hence for $200 \leq h \leq 2000$ we plot their aggregate number $\sum_{k=200}^{h} D_i(k)$ in Figure 10.1. The values $D_i(h)$ for $200 \leq h \leq 2000$ may be estimated from Figure 10.1, too.

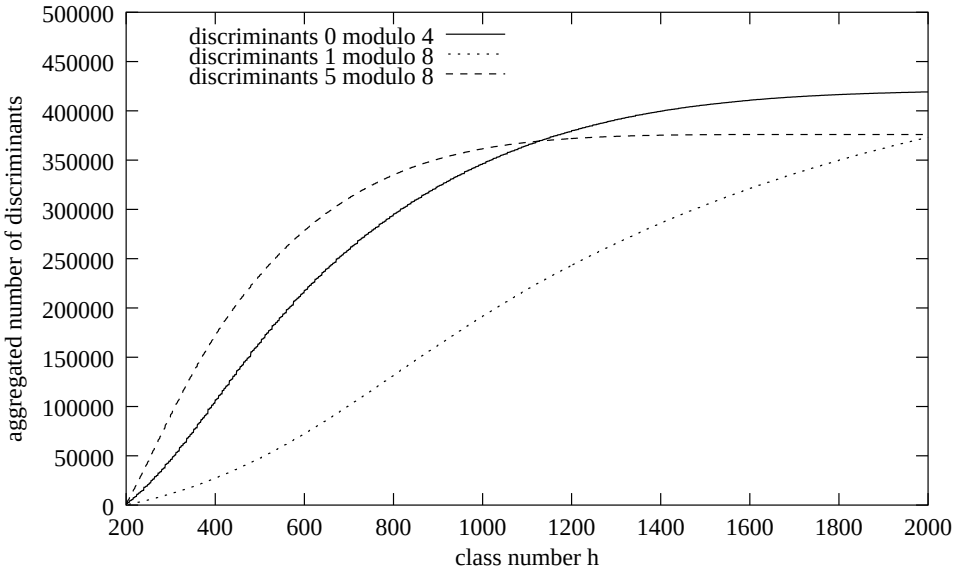


Figure 10.1: Number of different elliptic curves in the lift-attack model: Aggregate numbers $\sum_{k=200}^{h} D_i(k)$ for $i \in \{0, 1, 5\}$ and $200 \leq h \leq 2000$.

We conclude that there are sufficiently many fundamental discriminants inducing elliptic curves for use in cryptography. For instance, we have

$$\sum_{k=200}^{2000} D_0(k) = 419201 \,, \qquad \sum_{k=200}^{2000} D_1(k) = 373051 \,, \qquad \sum_{k=200}^{2000} D_5(k) = 376046 \,.$$

10.2.2 Number of Curves in the j-attack model

In this section we determine the number of different elliptic curves defined over a finite prime field in the j-attack model. This model assumes the existence of a polynomial time reduction of the ECDLP for curves of identical j-invariants. More precisely, two elliptic curves over $\mathbb{F}_p$ are considered as different if and only if their j-invariants are different. Hence, for a

given elliptic curve E over $\mathbb{F}_p$ all $\mathbb{F}_p$-isomorphic and all $\mathbb{F}_p$-twisted curves are considered as essentially the same one.

In case of $\mathbb{F}_p$-isomorphic curves the polynomial time reduction is easy to see. Let $E = (a, b)$ be an elliptic curve defined over $\mathbb{F}_p$, and assume the ECDLP to be easy in $E(\mathbb{F}_p)$. In addition, assume $E' = (a', b')$ to be $\mathbb{F}_p$-isomorphic to E. Theorem 2.3.21 shows the existence of an element $\alpha \in \mathbb{F}_p^\times$ with $(a\alpha^4, b\alpha^6) = (a', b')$. If either $b = 0$ or $a' = 0$ we have $\alpha^4 = a'/a$ or $\alpha^6 = b'/b$, respectively. Otherwise $\alpha^2 = ab'/(a'b)$ follows. Hence, the computation of α is polynomial in $\log p$. The polynomial reduction of the ECDLP in $E'(\mathbb{F}_p)$ to the ECDLP in $E(\mathbb{F}_p)$ is shown in the following diagram.

$$\begin{array}{ccc} G' = (x'_0, y'_0) & \xrightarrow{[l]G'} & P' = (x'_1, y'_1) \\ \downarrow{\scriptstyle E'\to E} & & \downarrow{\scriptstyle E'\to E} \\ G = (x'_0\alpha^{-2}, y'_0\alpha^{-3}) & \xrightarrow{[l]G} & P = (x'_1\alpha^{-2}, y'_1\alpha^{-3}) \end{array}$$

Let us turn to $\mathbb{F}_p$-twisted curves. We are not aware of any polynomial time reduction in $\log p$ between $\mathbb{F}_p$-twisted curves. If $E' = (a', b')$ is $\mathbb{F}_p$-twisted to E, there exists a non-square $\beta \in \mathbb{F}_p^\times$ with $(a\beta^2, b\beta^3) = (a', b')$. If $a' \neq 0 \neq b$, which is in general the case, we have $\beta = ab'/(a'b)$. Hence, the computation of β is polynomial in $\log p$. However, there is no isomorphism $E' \to E$ which is defined over $\mathbb{F}_p$. Over $\mathbb{F}_{p^2}$ an isomorphism may be defined using a square root $\gamma \in \mathbb{F}_{p^2}$ of β. This is an obvious consequence of $(a\gamma^4, b\gamma^6) = (a', b')$. The polynomial time reduction of the ECDLP in $E'(\mathbb{F}_{p^2})$ to the ECDLP in $E(\mathbb{F}_{p^2})$ is shown in the following diagram.

$$\begin{array}{ccc} G' = (x'_0, y'_0) & \xrightarrow{[l]G'} & P' = (x'_1, y'_1) \\ \downarrow{\scriptstyle E'\to E} & & \downarrow{\scriptstyle E'\to E} \\ G = (x'_0\gamma^{-2}, y'_0\gamma^{-3}) & \xrightarrow{[l]G} & P = (x'_1\gamma^{-2}, y'_1\gamma^{-3}) \end{array}$$

We remark that from an attacker's point of view the assumptions on the reduction for twisted curves are rather optimistic. Hence concerning security the j-attack model is very strong.

We next turn to the number of different cryptographically strong elliptic curves in the j-attack model. We fix a discriminant Δ. Then the number of such curves is essentially equal to the number of primes p of the form $(t^2 - \Delta y^2)/4$ such that either $p + 1 - t$ or $p + 1 + t$ is nearly prime. However, as discussed in Chapter 4 we are only able to determine this number in the case $\Delta \equiv 5 \bmod 8$. Nevertheless, we assume that in case of $\Delta \equiv 0 \bmod 4$ and $\Delta \equiv 1 \bmod 8$ the number of curves is of same order of magnitude, respectively.

In order to derive a formula for the number of different cryptographically strong elliptic curves of discriminant Δ in the j-attack model, let $\Delta \equiv 5 \bmod 8$ be fundamental and not divisible by 3. Furthermore, let $\mathcal{O}$ denote the maximal order of discriminant Δ. We are only interested in primes p suitable for the CM-method such that either $p + 1 - t$ or $p + 1 + t$ is prime, too. As discussed in Section 4.3 this is equivalent to the existence of a generalized twin prime $(\pi, \pi + 1) \in \mathcal{O}^2$. We write $T(b, \Delta)$ for the number of twin primes in $\mathcal{O}$ with $\lfloor \log_2 \pi\overline{\pi} \rfloor + 1 = b$. In addition, as $|t| < 2\sqrt{p}$, the probability of $\lfloor \log_2 \pi\overline{\pi} \rfloor \neq \lfloor \log_2 (\pi + 1)(\overline{\pi} + 1) \rfloor$ is negligible. Hence we may assume that $T(b, \Delta)$ is the number of primes p of bitlength b such that $p+1+t$ is prime of bitlength b, too.

From Section 4.3 we know that $T(b, \Delta)$ is equal to the numerator in Equation (4.8), that is

$$T(b, \Delta) = \frac{P_N(\Delta)\sqrt{|\Delta|}}{2\pi h^2} \int_{2^{b-1}}^{2^b - 1} \frac{dy}{(\log y)^2} \; . \tag{10.2}$$

In Lemma 4.3.2 we have shown that $C_1 \leq P_N(\Delta) \leq C_2$ with two (absolute) constants C_1 and C_2. In addition, making use of the asymptotic formula $\sqrt{|\Delta|} = O(h)$ of Theorem 2.1.17 and the approximation of the integral by $O(2^b/(\log 2^b)^2)$ we deduce $T(b, \Delta) = O(2^b/(b^2 \cdot h))$. Thus for fixed h the number of different elliptic curves in the j-attack model grows exponentially in b. This behavior is well demonstrated by Figure 10.2. We choose $\Delta = -125579$, as Δ is the largest fundamental discriminant congruent 5 modulo 8 of class number 200. Figure 10.2 shows $\log_{10}(T(b, -125579))$ as a function of b.

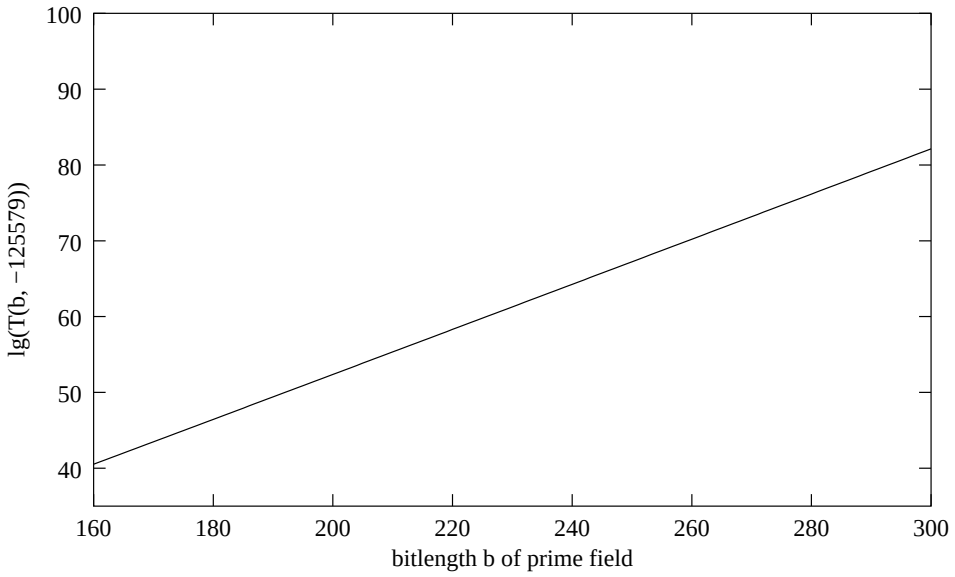


Figure 10.2: Number of different elliptic curves in the j-attack model: $\log_{10}(T(b, -125579))$ as a function of b for $160 \leq b \leq 300$.

Although Equation (10.2) bases on conjectures of Gross and Smith [GS00] and our interpretation of their claims, our practical tests shown in Figure 4.4 indicate that (10.2) is correct.

We conclude that in the j-attack model the number of different elliptic curves for use in cryptography is very large. For illustration, assume that there are $6 \cdot 10^9$ people living on earth. We further claim that we are only interested in an elliptic curve of prime order defined over a 160-bit field and having an endomorphism ring of discriminant -125579. In this case there are $3.3587189 \cdot 10^{40}$ generalized twin primes π such that $\pi\overline{\pi} \in [2^{159}, 2^{160} - 1]$. Hence each person gets $5.5978645 \cdot 10^{30}$ different curves, and neither curve is chosen twice. Thus, even in this very special case, the set of curves to choose from is sufficiently large.

Part III

Performance & Extensions of `cryptoCurve`

and

the Library **gec**

Chapter 11

Performance and Extensions of `cryptoCurve`

This chapter first presents in Section 11.1 sample running times of our generating algorithm `cryptoCurve` in practice. We measure its performance for various security levels, that is for different input r_0, k_0, and h_0. We show that the algorithm is able to generate an elliptic curve suitable for use in cryptography in about 12 seconds on the Pentium III without any precomputation. In addition, we will show that we are able to compute the coefficients of an elliptic curve having an endomorphism ring of class number 15000 in about 24 minutes on the Pentium III using a precomputed class polynomial.

Next, in Section 11.2 we extend our generating algorithm to find an elliptic curve such that both the curve and a twist of this curve are of prime order and suitable for use in cryptography. An application of this extension is a pseudo random number generator using elliptic curves, as described in [Kal86]. Finally, Section 11.3 describes an algorithm to find an elliptic curve of prime order over an Optimal Extension Field suitable for use in cryptography. This algorithm uses our ideas published in [Bai01c] and [Bai01d].

11.1 Practical Performance of `cryptoCurve`

This section provides sample running times of our algorithm `cryptoCurve` in practice. All timings are measured on the Pentium III. In addition, we refer to the data published in [BB00]; these timings come from the SUN UltraSPARC-IIi. All our timings show that our algorithm is very efficient in practice. First, in Section 11.1.1 we present running times of our algorithm `cryptoCurve`(r_0, k_0, h_0) without any precomputation and using the Fixed Discriminant Approach of Chapter 4. It turns out that `cryptoCurve` is very fast for $h_0 \in \{200; 1000\}$. For example, if $r_0 = 2^{159}$, $k_0 = 4$ and $h_0 = 200$ the running time of `cryptoCurve`(r_0, k_0, h_0) is less than 12 seconds on the Pentium III.

Second, in Section 11.1.2 we describe our database classPolynomials, which stores class polynomials W for various discriminants. We provide timings for the computation of a class polynomial W if a discriminant of large class number is chosen (that is a class number in the range $[3000; 15000]$). Furthermore, we give sample timings of `cryptoCurve` using the database classPolynomials. Again, we use the Fixed Discriminant Approach of Chapter 4.

For example, the running time to compute an elliptic curve having an endomorphism ring of class number 15000 takes less than 24 minutes on the Pentium III. Hence our algorithm is applicable even in case of discriminants having a very large class number.

Finally, we refer to Chapter B of the appendix for sample curves of the following sections.

11.1.1 Performance without Using Precomputed Class Polynomials

We present running times of our algorithm `cryptoCurve`(r_0, k_0, h_0) without precomputation of a class polynomial. All timings in this section come from the Fixed Discriminant Approach of Chapter 4.

We first turn to timings of `cryptoCurve`(r_0, k_0, h_0) for $r_0 = 2^{159}$, $k_0 = 4$, and $200 \leq h_0 \leq 1000$. The timings are given in Table 11.1. As the cofactor is allowed to be at most 4, we choose $\Delta \equiv 1 \bmod 8$ and $3 \nmid \Delta$. Thus our subalgorithm `findRoot` computes the class polynomial W. In addition, for given h_0 we choose Δ to be maximal and fundamental with $h(\Delta) = h_0$, $\Delta \equiv 1 \bmod 8$, and $3 \nmid \Delta$. From Table 11.1 we conclude that even for a discriminant of class number 1000 we are able to compute an elliptic curve in about 80 seconds. Furthermore, we see that most of the time is spent to compute a root of W mod p.

Δ	h_0	time of `computeClassInvariants` and `computeClassPolynomial`	time of `find_root`	time of `cryptoCurve`
-21311	200	0.62	10.39	11.29
-30551	250	0.87	11.81	13.18
-34271	300	1.18	20.96	22.84
-47759	350	1.58	22.26	24.17
-67031	400	2.2	24.07	26.72
-75599	450	2.6	25.35	28.41
-96599	500	3.45	26.67	30.66
-148511	600	5.33	45.8	51.43
-185471	700	7.23	50.13	57.7
-233999	800	10.12	52.96	63.68
-299519	900	13.94	56.75	71.1
-412079	1000	19.47	59.63	79.81

Table 11.1: Running times of `cryptoCurve`$(2^{159}, 4, h_0)$ on the Pentium III for $200 \leq h_0 \leq 1000$. All timings are given in seconds. For given h_0, the discriminant Δ is maximal and fundamental with $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$ and $h(\Delta) = h_0$. The class polynomial W is computed during the algorithm.

In Section A.4 of the appendix we provide additional sample running times in case of $k_0 = 4$. For instance, Table A.1 on Page 204 lists timings of `cryptoCurve`$(r_0, 4, 200)$ on the Pentium III for $r_0 = 2^b$, $159 \leq b \leq 499$. The discriminant is $\Delta = -21311$; Δ is the maximal fundamental discriminant of class number 200, which is congruent 1 modulo 8 and not divisible by 3. Table A.1 shows that `cryptoCurve`$(2^{499}, 4, 200)$ terminates successfully after 2 minutes and 15 seconds. Furthermore, Table A.2 on Page 205 provides running times if the class number of the discriminant is 500.

Next, we present running times to find elliptic curves of prime order. Table 11.2 lists timings

of `cryptoCurve`$(2^{159}, 1, h_0)$ for $200 \leq h_0 \leq 1000$. As the elliptic curve is of prime order, we have to ensure $\Delta \equiv 5 \bmod 8$. In addition, we choose Δ to be not divisible by 3. Hence in this case our subalgorithm `findRoot` uses the class polynomial G. As above, if h_0 is given, Δ is chosen to be maximal and fundamental with $h(\Delta) = h_0$, $\Delta \equiv 5 \bmod 8$ and $3 \nmid \Delta$. We conclude from Table 11.2 that we are able to generate an elliptic curve of prime order suitable for use in cryptography in less than 30 seconds. In addition, if we choose a discriminant of class number 500 the computation takes about 7 minutes.

In contrast to the timings in case of $k_0 = 4$, most of the time is spent to compute the class polynomial G. Roughly speaking, the proportion of this step increases with growing class number. If the class number is chosen to be 1000, `cryptoCurve` becomes rather slow. In this case the running time to generate an elliptic curve of prime order is about 73 minutes; 98.6% of this time is spent to compute the polynomial G.

Δ	h_0	time of `computeClassInvariants` and `computeClassPolynomial`	time of `find_root`	time of `cryptoCurve`
-125579	200	18.52	9.61	28.7
-184091	250	39.28	10.96	50.75
-223739	300	61.06	20.14	81.76
-294971	350	105.31	21.9	127.87
-428819	400	182.32	23.3	206.42
-539579	450	284.11	23.94	308.62
-742979	500	405.03	26.05	432.02
-834539	600	654.55	45.76	701.37
-1166051	700	1189.32	48.39	1240.22
-1390091	800	1859.46	52	1912.45
-1908539	900	2739.03	53.97	2794.28
-2656979	1000	4314.21	57.75	4374.04

Table 11.2: Running times of `cryptoCurve`$(2^{159}, 1, h_0)$ on the Pentium III for $200 \leq h_0 \leq 1000$. All timings are given in seconds. For given h_0, the discriminant Δ is maximal and fundamental with $\Delta \equiv 5 \bmod 8$, $3 \nmid \Delta$ and $h(\Delta) = h_0$. The class polynomial G is computed during the algorithm.

Furthermore, Section A.4 of the appendix provides additional sample run times to compute curves of prime order. Table A.3 on Page 206 gives timings of `cryptoCurve`$(r_0, 1, 200)$ on the Pentium III for $r_0 = 2^b$, $159 \leq b \leq 499$. The discriminant is $\Delta = -125579$; Δ is the maximal fundamental discriminant of class number 200, which is congruent 5 modulo 8 and not divisible by 3. We conclude from Table A.3 that we are able to generate an elliptic curve of prime order over a 500-bit field and having an endomorphism ring of class number 200 in about 2 minutes and 30 seconds.

11.1.2 Performance Using Precomputed Class Polynomials

In this section we present running times of our algorithm `cryptoCurve` if the class polynomial has been computed in advance. We first describe our database classPolynomials, which stores class polynomials W. We then give timings to compute a class polynomial W for discriminants

of class numbers in $[3000; 15000]$. Again, all timings are measured on the Pentium III; in `cryptoCurve` we make use of the Fixed Discriminant Approach of Chapter 4.

Let us first turn to our database classPolynomials. It stores class polynomials W for various discriminants. Table 11.3 gives an overview of the database. In all, classPolynomials comprises 178897 class polynomials W. The amount of storage is about 23 GByte. For instance, the class polynomial W according to the discriminant $\Delta = -55222439$ of class number 15000 uses 45.0 MByte memory. Its largest coefficient (with respect to the absolute value) is w_{3862}, which is the coefficient of X^{3862}. This coefficient may be found in Section A.5 of the appendix. Its bitlength and decimal length is 11892 and 3580, respectively.

class number h	additional condition on Δ	number of polynomials
$200 \leq h \leq 799$	$\Delta > -6000000$	143526
$800 \leq h \leq 1499$	first 50 discriminants	35000
$1500 \leq h \leq 4900$, $100 \mid h$	first 10 discriminants	350
$5000 \leq h \leq 15000$, $500 \mid h$	first discriminant	21

Table 11.3: Overview of database classPolynomials. We only consider discriminants Δ which are congruent 1 modulo 8 and not divisible by 3. For either discriminant classPolynomials stores the corresponding class polynomial W.

We next turn to the running time to compute a class polynomial W of large degree. If h denotes a class number, we choose the discriminant Δ to be maximal and fundamental with $h(\Delta) = h$, $\Delta \equiv 1 \bmod 8$, and $3 \nmid \Delta$. Figure 11.1 plots the CPU-timings on the Pentium III as a function of the class number h. While the computation of a class polynomial W for $h = 3000$ only takes about 8 and a half minutes, the CPU-time in case of $h = 15000$ increases to 47.9 hours.

In Table A.4 on Page 208 we present CPU-timings of `cryptoCurve`$(2^{159}, 4, h_0)$ on the Pentium III, if $3000 \leq h_0 \leq 15000$ and if we make use of our database classPolynomials. We deduce from this table that `cryptoCurve`$(2^{159}, 4, 3000)$ terminates successfully after 4 and a half minutes. Furthermore, if we choose $h_0 = 15000$ the running time is about 23 and a half minutes. This proves that our algorithm is applicable even for very large class numbers, that is class numbers ≥ 10000.

In addition to the run time benefit when using class polynomials stored in the database classPolynomials we mention a further advantage. All polynomials in classPolynomials have passed successfully the probabilistic correctness test `isWeberPolynomial` explained at Page 138. Hence we are convinced that all polynomials in classPolynomials are correct.

11.2 Generating Curves and Twists of Prime Order

In this section we present a first extension of our algorithm `cryptoCurve`. We show how to efficiently find an elliptic curve E defined over $\mathbb{F}_p$ such that both $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_p)$ are cryptographically strong, where E' denotes a twisted curve of E over $\mathbb{F}_p$. We call our algorithm `twistedPair`.

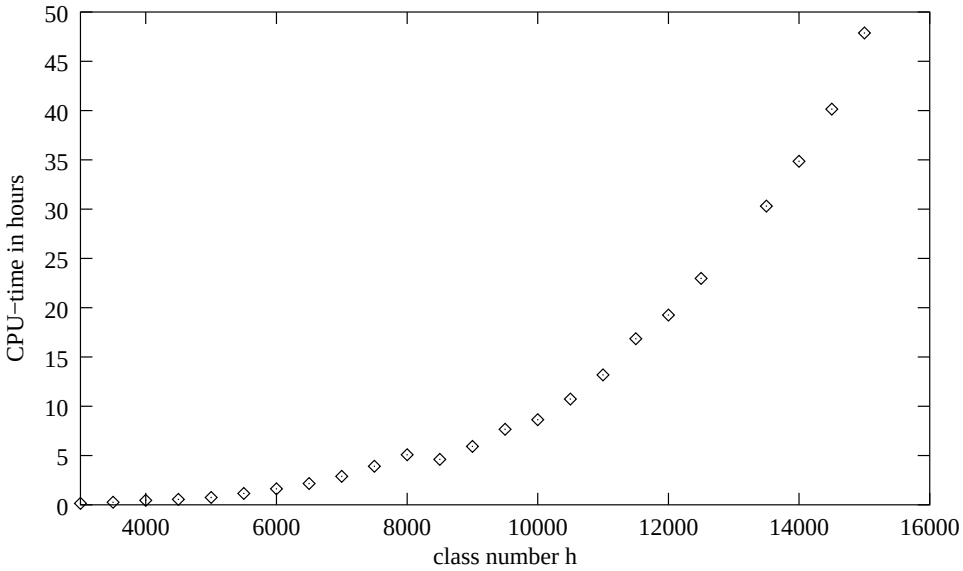


Figure 11.1: Timings on the Pentium III to compute a class polynomial W of large degree. For each class number h with $3000 \leq h \leq 15000$, $500 \mid h$, the maximal fundamental discriminant with $h(\Delta) = h$, $\Delta \equiv 1 \bmod 8$, and $3 \nmid \Delta$ is chosen. The running time is given in hours and plotted as a function of h.

`twistedPair` may be used to implement a pseudo random bit generator using elliptic curves. Such a generator was first proposed by B. Kaliski in [Kal86] and [Kal88]. The security of the bit generator bases on the ECDLP. However, to put Kaliski's proposal into practice, one has to find an elliptic curve E defined over $\mathbb{F}_p$ such that the ECDLP is intractable in both $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_p)$. In addition, one has to know generating points of both groups. In general these points are found using the Weil-pairing, and this is a non-trivial task.

In order to avoid the Weil-pairing, we propose to apply elliptic curves of prime order. More precisely, we adapt our main algorithm `cryptoCurve` to find an elliptic curve E such that both $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_p)$ are of prime order and cryptographically strong. Hence, every non-zero point generates the set of rational points over $\mathbb{F}_p$. Thus we get the group structure for free. In addition, as the cofactor is 1 arithmetic on the curves for given security level is rather efficient. Furthermore, we present running times demonstrating the efficiency of our generating algorithm in practice. We are not aware of any further algorithm finding such curve pairs.

Definition 11.2.1 *Let $p \geq 5$ be prime and E be an elliptic curve over $\mathbb{F}_p$. Furthermore, let E' be $\mathbb{F}_p$-twisted to E. If both $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_p)$ are of prime order and cryptographically strong, we call the pair (E, E') a* twisted pair.

We describe our algorithm `twistedPair`$(\Delta, [i_0, i_1])$. Its input is an imaginary quadratic discriminant Δ and an interval $[i_0, i_1] \subset \mathbb{N}$. The algorithm outputs a prime p and a twisted pair (E, E') such that $|E(\mathbb{F}_p)|, |E'(\mathbb{F}_p)| \in [i_0, i_1]$ and $\mathrm{End}(E) = \mathrm{End}(E') = \mathcal{O}_\Delta$.

Our requirement that both $|E(\mathbb{F}_p)|$ and $|E'(\mathbb{F}_p)|$ are of prime order imposes some restrictions on the choice of Δ. First, as discussed in Section 4.3, we have to ensure $\Delta \equiv 5 \bmod 8$. Second, from the tables in Section A.1.2 we deduce the condition $\Delta \equiv 2 \bmod 3$. Thus we have to take care of the congruence $\Delta \equiv 5 \bmod 24$.

We discuss the algorithm in detail in what follows. `twistedPair` first invokes algorithm `findPrimeTwistedPair`$(\Delta, [i_0, i_1])$. Given the discriminant Δ and the interval $[i_0, i_1]$, this method returns primes p, r, and r' having the following three properties: First, the norm equation $4p = t^2 - \Delta y^2$ has a solution $(t, y) \in \mathbb{Z}^2$. Second, the primes r and r' are of the form $p + 1 - t$ and $p + 1 + t$, respectively. Third, r respectively r' are orders of $\mathbb{F}_p$-twisted cryptographically strong elliptic curves over $\mathbb{F}_p$ having $\mathcal{O}_\Delta$ as endomorphism ring. This task is similar to the problem discussed in Section 4.3.2. Hence, `findPrimeTwistedPair` is alike to algorithm `findPrime5Mod8`. However, we have to consider some changes.

m	$t \bmod m$	$y \bmod m$
2	1	1
3	0, 1, 2	1, 2
5	1, 4	0
7	1, 3, 4, 6	0
11	1, 3, 4, 5, 6, 7, 8, 10	0

Table 11.4: Congruence conditions on t and y for $p + 1 - t$ and $p + 1 + t$ to be prime. Δ is assumed to be congruent 5 mod 24.

First, as in Section 4.3.2, we collect congruence conditions on $(t \bmod m, y \bmod m)$ for small moduli m. We evaluate the moduli $m \in \{2, 3, 5, 7, 11\}$. Again, to avoid distinguishing a multitude of cases, the initial chosen value of y leaves unchanged. We refer to Section 4.3.2 for the derivation of the conditions and summarize the results in Table 11.4.

c	(t_5, t_7, t_{11})
19	(1, 1, 3), (1, 3, 3)
14	(1, 1, 1), (1, 1, 10), (1, 3, 1), (1, 4, 4), (1, 6, 4), (1, 6, 6)
11	(4, 3, 1), (4, 3, 10)
10	(1, 4, 6), (4, 4, 7)
9	(1, 4, 1), (1, 4, 3), (1, 4, 5), (1, 4, 7), (1, 4, 8), (1, 4, 10), (1, 6, 1), (1, 6, 3), (1, 6, 5), (1, 6, 7), (1, 6, 8), (1, 6, 10)
6	(4, 3, 3), (4, 6, 7)
5	(1, 1, 4), (1, 1, 5), (1, 1, 6), (1, 1, 7), (1, 1, 8), (1, 3, 5), (4, 1, 7), (4, 3, 4), (4, 3, 5), (4, 3, 6), (4, 3, 7), (4, 3, 8)
4	(1, 3, 4), (1, 3, 6), (1, 3, 7), (1, 3, 8), (1, 3, 10)
1	(4, 1, 1), (4, 1, 3), (4, 1, 4), (4, 1, 5), (4, 1, 6), (4, 1, 8), (4, 1, 10), (4, 4, 1), (4, 4, 3), (4, 4, 4), (4, 4, 5), (4, 4, 6), (4, 4, 8), (4, 4, 10), (4, 6, 1), (4, 6, 3), (4, 6, 4), (4, 6, 5), (4, 6, 6), (4, 6, 8), (4, 6, 10)

Table 11.5: Minimal values c such that both (t_5, t_7, t_{11}) and $(t_5 + 2c \bmod 5, t_7 + 2c \bmod 7, t_{11} + 2c \bmod 11)$ respect the requirements of Table 11.4.

Next, assume (t, y) to be a pair respecting the requirements of Table 11.4. We explain how we modify (t, y) such that the modified pair satisfies the congruence conditions, too. To keep track of the residues of t modulo the primes m of Table 11.4 we introduce variables t_5, t_7, and t_{11}. They are simply defined as t mod 5, t mod 7, and t mod 11, respectively. We remark that we do not have to consider requirements for t mod 3. However, when modifying t we have to ensure that t keeps odd. Hence we successively increase t by 2 until the modified residues t_5, t_7, and t_{11} all turn up in Table 11.4, too. Hence, we determine the minimal value c such that all t_5, t_7, t_{11}, $t_5 + 2c$ mod 5, $t_7 + 2c$ mod 7, and $t_{11} + 2c$ mod 11 appear in Table 11.4. The values of c are listed in Table 11.5.

Our function `cyclesTwistedPair`(t_5, t_7, t_{11}) implements the requirements of Table 11.5. Given (t_5, t_7, t_{11}) it returns the corresponding c from Table 11.5.

We remark that the primes p returned by algorithm `findPrimeTwistedPair` are K-uniformly distributed in $[i_0, i_0 + K \cdot \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor]$. Hence, if K does not divide $i_1 - i_0 + 1$, we cut off the end of the interval. In our implementation, we set $K = 2^{40}$. Furthermore, according to our tests the choice $M = 24$ and $T = 2500$ seems to be optimal.

Once the primes p, r, and r' are determined, `twistedPair` proceeds similarly as our main algorithm `cryptoCurve`. As we have $\Delta \not\equiv 1 \bmod 8$ and $3 \nmid \Delta$ the subalgorithm `findRoot` makes use of the class polynomial G to generate the ring class field. It returns the j-invariant of elliptic curves over $\mathbb{F}_p$ of order r and r', respectively. Using this j-invariant, we determine the coefficients of the curves. Finally, by trial and error, we assign the appropriate cardinality to either curve.

Finally, we come up with sample practical running times. In Table 11.6 we list timings measured on the Pentium III. For each class number the chosen discriminant is maximal with $\Delta \equiv 5 \bmod 24$. In addition, all discriminants are fundamental. The timings are given in seconds. The running time for `findPrimeTwistedPair` is the average time of 100 tests. The total time of `twistedPair` is roughly the sum of the given timings.

We conclude that we are able to find a twisted pair respecting all requirements of Section 2.3.5 in about 40 seconds on an average PC. Furthermore, even for discriminants of class number 500 we generate a twisted pair in less than 10 minutes. Hence, `twistedPair` is a very efficient algorithm to generate twisted pairs. We are not aware of any comparable algorithm.

Δ	h	Time of `findPrimeTwistedPair`	Time to compute class polynomial G	Time of `find_root`
-356131	200	2.1096	25.73	9.73
-467011	250	2.0866	45.58	11.14
-881011	300	2.8276	95.52	20.31
-1190251	350	2.1262	171.81	21.41
-1531339	400	2.4975	248.51	23.01
-1825291	450	2.1254	356.62	24.49
-2299699	500	2.3858	545.13	26.16

Table 11.6: Running times of `twistedPair`$(\Delta, [2^{159}, 2^{160} - 1])$ on the Pentium III. All timings are given in seconds. The running time of `twistedPair` is roughly the sum of the given subalgorithms.

Algorithm 11.1: `findPrimeTwistedPair`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 5 \bmod 24$.
An interval $[i_0, i_1] \subset \mathbb{N}$ with $i_1 - i_0 + 1 \geq 2^{159}$.
Output: A prime $p \in [i_0, i_1]$.
Primes $r \geq i_0$ and $r' \geq i_0$ being orders of cryptographically strong $\mathbb{F}_p$-twisted elliptic curves E and E' over $\mathbb{F}_p$ with $\mathrm{End}(E) = \mathrm{End}(E') = \mathcal{O}_\Delta$.

$K \leftarrow 2^{40}$; $m \leftarrow \lfloor \frac{i_1 - i_0 + 1}{K} \rfloor$; $T \leftarrow 2500$; $M \leftarrow 24$;
$m_R \leftarrow 50$; //set the number of Miller-Rabin tests to 50
while `true` **do**
  $s \in_R \{0, \ldots K-1\}$; $v \leftarrow i_0 + sm$; $w \leftarrow v + m - 1$; //choose a random subinterval
  $\tau \in_R \{1, ..., \lfloor 2\sqrt{v} \rfloor\}$; $\mu \leftarrow \left\lceil \sqrt{(4v - \tau^2)/|\Delta|} \right\rceil$;
  $t \leftarrow \min\{x \geq \tau : x \equiv 1 \bmod 770\}$;
  $y \leftarrow \min\{x \geq \mu : x \equiv 385 \bmod 770, 3 \nmid x\}$;
  $p \leftarrow (t^2 - \Delta y^2)/4$;
  $c \leftarrow 0$; $c_r \leftarrow 0$; $j \leftarrow 0$;
  $t_5 \leftarrow 1$; $t_7 \leftarrow 1$; $t_{11} \leftarrow 1$;
  for $i = 0$ to $M - 5$ **do**
    $p[i] \leftarrow p \bmod p_{i+5}$; //initialize $p[\,]$, i.e. $p \bmod 13, \ldots, p \bmod 97$
    $t[i] \leftarrow t \bmod p_{i+5}$; //initialize $t[\,]$, i.e. $t \bmod 13, \ldots, t \bmod 97$
  while $j < T$ **do**
    if $p[i] \neq 0$ AND $p[i]+1-t[i] \bmod p_{i+5} \neq 0$ AND $p[i]+1+t[i] \bmod p_{i+5} \neq 0$ for all $0 \leq i \leq M-5$ **then**
      $p \leftarrow p + c_r t + c_r^2$; $t \leftarrow t + 2c_r$; $c_r \leftarrow 0$; $j \leftarrow j + 1$;
      if `isPrime`$(p, m_R) =$ `true` AND `isPrime`$(p+1-t, m_R) =$ `true` AND `isPrime`$(p+1+t, m_R) =$ `true` **then**
        if `isStrong`$(i_0, 1, p, p+1-t) \neq 0$ AND `isStrong`$(i_0, 1, p, p+1+t) \neq 0$ **then**
          return($p, p+1-t, p+1+t$);
    $c \leftarrow$ `cyclesTwistedPair`(t_5, t_7, t_{11}); $c_r \leftarrow c_r + c$;
    $t_5 \leftarrow t_5 + 2c \bmod 5$; $t_7 \leftarrow t_7 + 2c \bmod 7$; $t_{11} \leftarrow t_{11} + 2c \bmod 11$;
    for $i = 0$ to $M - 5$ **do**
      $p[i] \leftarrow p[i] + ct[i] + c^2 \bmod p_{i+5}$; $t[i] \leftarrow t[i] + 2c \bmod p_{i+5}$;

In Section B.4 we list some sample twisted pairs over finite prime fields $\mathbb{F}_p$. First, in Section B.4.1, we present examples defined over a field of minimal bitlength, that is p is of bitlength 160. The class number of the endomorphism ring of the curves in Section B.4.1 varies from 200 to 500. In addition, in Section B.4.2 we consider fields of bitlength up to 500. In Section B.4.2 the discriminant of the endomorphism ring is chosen to be -356131. It is the maximal fundamental discriminant of class number 200 which is congruent 5 modulo 24.

In order to ensure an efficient implementation of the curve arithmetic, all twisted pairs in Section B.4 are of the form $((-3, b), (-3, b'))$ for some $b, b' \in \mathbb{F}_p$. However, it turns out that the existence of a twisted pair of the form $((-3, b), (-3, b'))$ implies $p \equiv 3 \bmod 4$, as we now see.

Proposition 11.2.2 *Let $p > 3$ be prime and let $((-3, b), (-3, b'))$ be a twisted pair for some $b, b' \in \mathbb{F}_p$. Then $p \equiv 3 \bmod 4$.*

Proof: Let $((-3, b), (-3, b'))$ be a twisted pair. According to Definition 2.3.21 there exists a

Algorithm 11.2: `twistedPair`$(\Delta, [i_0, i_1])$

Input: A discriminant $\Delta \in \mathbb{Z}_{<0}$, $\Delta \equiv 5 \bmod 24$.
An interval $[i_0, i_1] \subset \mathbb{N}$ with $i_1 - i_0 + 1 \geq 2^{159}$.
Output: A prime $p \in [i_0, i_1]$ of the form $(t^2 - \Delta y^2)/4$.
Primes $r \geq i_0$ and $r' \geq i_0$ of the form $p + 1 - t$ and $p + 1 + t$, respectively.
$\mathbb{F}_p$-twisted cryptographically strong elliptic curves E and E' over $\mathbb{F}_p$ with $|E(\mathbb{F}_p)| = r$, $|E'(\mathbb{F}_p)| = r'$, and $\mathrm{End}(E) = \mathrm{End}(E') = \mathcal{O}_\Delta$.
Generating points G and G' of $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_p)$, respectively.

$(p, r, r') \leftarrow$ `findPrimeTwistedPair`$(\Delta, [i_0, i_1])$;
$j_p \leftarrow$ `findRoot`(Δ, p);
Select a quadratic non-residue $s_p \bmod p$; $\kappa_p \leftarrow j_p/(1728 - j_p)$; $a_p \leftarrow 3\kappa_p$; $b_p \leftarrow 2\kappa_p$;
$E_1 \leftarrow (a_p, b_p)$; $E_2 \leftarrow (a_p s_p^2, b_p s_p^3)$;
$G_1 \in_R E_1(\mathbb{F}_p) \setminus \{O\}$; $G_2 \in_R E_2(\mathbb{F}_p) \setminus \{O\}$;
if $rG_1 = O$ **then**
  return $(p, E_1, G_1, r, E_2, G_2, r')$;
else
  return $(p, E_2, G_2, r, E_1, G_1, r')$;

non-square $\beta \in \mathbb{F}_p^\times$ with $-3 = -3\beta^2$. As we assume $p \geq 5$, we have $\beta^2 = 1$. This equation has the two different solutions $\{1; -1\}$. As 1 is obviously a square, we deduce $\beta = -1$. Thus -1 is a non-square in $\mathbb{F}_p^\times$. However, -1 is a non-square in $\mathbb{F}_p^\times$ if and only if $p \equiv 3 \bmod 4$. □

11.3 Elliptic Curves of Prime Order over Optimal Extension Fields Suitable for Use in Cryptography

We present an algorithm for generating elliptic curves of prime order over Optimal Extension Fields suitable for use in cryptography. We call our algorithm `oefCurve`. Furthermore, we demonstrate the efficiency of the algorithm in practice by giving practical running times. In addition, we present statistics on the number of cryptographically strong elliptic curves of prime order for Optimal Extension Fields of cardinality $(2^{32} + c)^5$ with $c < 0$. We conclude that there are sufficiently many curves in this case.

As of today, only two families of finite fields have found consideration for elliptic curve cryptography in practice: Finite fields of characteristic 2 and finite prime fields of large characteristic. Algorithms to find elliptic curves for use in cryptography are well known for both families of fields. Recently, a new type of finite fields was proposed for use in practice: Optimal Extension Fields ([BP98], [BP01]). Optimal Extension Fields consider the hardware in use (i.e. the word size of the processor) and thus yield an efficient way of implementing finite field arithmetic, especially the inversion. As the inversion is the most time-consuming step for adding points on elliptic curves over finite fields, Optimal Extension Fields have the potential to be considered as a third family of finite fields for elliptic curve cryptography.

Processors of word size 32-bit play a crucial role in practice. Hence, we will show that our algorithm is very fast in this case. Let p be a 32-bit prime with $2^{32} - p < 2^{16}$. Our algorithm finds a cryptographically strong elliptic curve of prime order over an Optimal Extension Field

$\mathbb{F}_{p^5}$ in about 22 seconds using an ordinary PC. In addition, we present data on the number of suitable elliptic curves over Optimal Extension Fields of the form $\mathbb{F}_{p^5}$. We conclude that, for fields of this form, their quantity is sufficiently large.

This section is organized as follows: First, in Section 11.3.1 we review the basic terms in the framework of Optimal Extension Fields. Next we present our generating algorithm in Section 11.3.2. Finally, in Section 11.3.3 we give running times of our implementation, and discuss statistics on the number of elliptic curves of prime order over fields of the form $\mathbb{F}_{p^5}$ with a 32-bit prime p.

11.3.1 Elliptic Curves over Optimal Extension Fields

We review the definition and some properties of Optimal Extension Fields. Let us first turn to its definition.

Definition 11.3.1 *Let c be a rational integer, and let $p = 2^n + c$ be prime with $n \in \mathbb{N}$. Furthermore, assume $|c| \leq \sqrt{2^n}$, and let $m \in \mathbb{N}$. If there is a $\omega \in \mathbb{F}_p$ such that the binomial $X^m - \omega$ is irreducible in $\mathbb{F}_p[X]$, then $\mathbb{F}_{p^m}$ is called an* Optimal Extension Field.

The basic idea of introducing Optimal Extension Fields is to adapt the arithmetic over finite extension fields to the hardware in use (see [BP98], [BP01]). For instance, when implementing an elliptic curve cryptosystem on a 32-bit processor, one may choose $n = 32$ and $c < 0$ such that $2^{32} + c$ is prime. Hence, the arithmetic in $\mathbb{F}_p$ fits in a word size. Furthermore, let ω be as in Definition 11.3.1. We represent $\mathbb{F}_{p^m}$ as the factor ring $\mathbb{F}_p[X]/(X^m - \omega)$ with respect to the polynomial basis $\{1, X, X^2, \ldots, X^{m-1}\}$. Thus in $\mathbb{F}_{p^m}$ the identity $X^m = \omega$ holds, yielding an easy reduction of X^k for $k \geq m$. In addition, all coefficients of a polynomial representative may be represented within one register of the processor.

Bailey and Paar [BP01] distinguish two special types of Optimal Extension Fields: First, if $|c| = 1$, the according Optimal Extension Field is called a *Type I* OEF. Second, if $X^m - 2$ is irreducible in $\mathbb{F}_p[X]$, they call the according field *Type II* OEF. In this thesis we do not make use of Type I OEFs.

In order to decide whether an irreducible binomial of degree m exists in $\mathbb{F}_p[X]$ we make use of the following theorem.

Theorem 11.3.2 *Let p and m be rational primes. For $\omega \in \mathbb{F}_p^\times$ the following properties are equivalent:*
1. *The binomial $X^m - \omega$ is irreducible in $\mathbb{F}_p[X]$.*
2. *m divides the order e of ω in $\mathbb{F}_p^\times$, but not $\frac{p-1}{e}$.*
3. *We have $m \mid p-1$ and $\omega^{\frac{p-1}{m}} \not\equiv 1 \bmod p$.*

Proof: The equivalence between 1 and 2 is a special case of theorem 3.75 in [BP01].
Let $p - 1 = m^{e_m} \cdot R$ with $(R, m) = 1$. Obviously, property 2 is equivalent to $e = m^{e_m} \cdot R'$ with $R' \mid R$. Furthermore, $\omega^{\frac{p-1}{m}} \not\equiv 1 \bmod p$ is equivalent to $e \nmid \frac{p-1}{m} = m^{e_m - 1} \cdot R$. If 2 holds we obviously have $e \nmid \frac{p-1}{m}$, hence 3 follows. Finally, assume 3. Hence we have $e \nmid \frac{p-1}{m} = m^{e_m - 1} \cdot R$. However, as e has to divide $p - 1$, e has to be of the form $m^{e_m} \cdot R'$ with some $R' \mid R$. □

Using the property that $\mathbb{F}_p^\times$ is a cyclic group, the following corollary is an easy consequence of property 3 in Theorem 11.3.2.

Corollary 11.3.3 *Let p and m be primes. There exists an irreducible binomial of degree m in $\mathbb{F}_p[X]$ if and only if $m \mid p-1$.*

In our investigation we focus on Optimal Extension Fields of the form p^5 with a 32-bit prime p. The reason for the choice $m = 5$ is twofold. Due to the theorem of Hasse (Theorem 2.3.18) we have $|E(\mathbb{F}_q)| \approx q$. Hence, in order to generate an elliptic curve of prime order r with $r \approx 2^{160}$ we have to ensure $m \geq 5$. Thus the choice $m = 5$ is minimal. Second, we restrict to extension fields of prime degree as some of our subalgorithms of Section 11.3.2 are very efficient in this case. However, the security implication of the Weil-descent ([GHS01]) on extension fields of degree 5 is not yet clear. For instance, in case of fields of the form $\mathbb{F}_{2^{155}}$ there are different statements. Smart argues in [Sma01] that given the current state of knowledge and technology elliptic curves over $\mathbb{F}_{2^{155}}$ are secure if they respect similar requirements as in Section 2.3.5. In contrast to Smart's statement is the investigation of Jacobson, Menezes, and Stein in [JMS01]. They showed that for an instance of an elliptic curve over $\mathbb{F}_{2^{155}}$ which resists all previously known attacks, an attack using the Weil-descent was successful. However, these investigations consider fields of characteristic 2. Nevertheless, the generalization of our approach to composite $m \geq 6$ is easy.

We are not aware of any further efficient algorithm to find an elliptic curve over an Optimal Extension Field of characteristic ≥ 5 respecting all these requirements. Although the Schoof-Elkies-Atkin (SEA) algorithm is of polynomial complexity in the bitlength of q for arbitrary finite fields $\mathbb{F}_q$ and efficiently implemented for Optimal Extension Fields, it turns out to be much slower in practice. The main reason is that we have to choose a number of curves and determine their cardinalities before finding a suitable one. Furthermore, the very efficient Satoh-algorithm for fields of characteristic 2 ([Sat99], [FGH01]) does not apply to Optimal Extension Fields.

11.3.2 The Algorithm `oefCurve`

In this section we present our generating algorithm `oefCurve`. The algorithm is listed at the end of this section. `oefCurve` makes use of the theory of complex multiplication. However, we have to extend the proceeding described in Chapter 3. Our aim is to find rational primes p and m, and an elliptic curve defined over $\mathbb{F}_{p^m}$, but not over $\mathbb{F}_p$. If E was defined over $\mathbb{F}_p$, then $E(\mathbb{F}_p)$ would be a subgroup of $E(\mathbb{F}_{p^m})$. Hence, for a 32-bit prime p, the curve would not respect the requirement $k \leq 4$.

We first have to find a prime power p^m and a discriminant Δ, such that the norm equation

$$t^2 - \Delta y^2 = 4p^m \tag{11.1}$$

has a solution $(t, y) \in \mathbb{Z}^2$, while the equation $t'^2 - \Delta y'^2 = 4p$ does not have a solution $(t', y') \in \mathbb{Z}$. If this is true, using complex multiplication (similar as described in Section 3), we find elliptic curves $E_{1,q}$ and $E_{2,q}$ over $\mathbb{F}_{p^m}$, both not defined over $\mathbb{F}_p$, with

$$|E_{1,q}(\mathbb{F}_{p^m})| = p^m + 1 - t, \quad |E_{2,q}(\mathbb{F}_{p^m})| = p^m + 1 + t \tag{11.2}$$

analogously as explained in Section 2.3.4.

As usual, let $H := H_\Delta$ be the ring class polynomial. Modulo p the polynomial H splits into irreducible factors of degree m, while it splits in $\mathbb{F}_{p^m}[X]$ into pairwise distinct linear factors. Let $j_q \in \mathbb{F}_{p^m}$ be a zero of H mod p. If $\Delta < -4$, we have $j_q \notin \{0; 1728\}$, and for any non-square $s_q \in \mathbb{F}_{p^m}$ we set as usual

$$\kappa_q = \frac{j_q}{1728 - j_q}, \quad (a_q, b_q) = (3\kappa_q, 2\kappa_q) \; . \tag{11.3}$$

Then we have

$$\{E_{1,q}, E_{2,q}\} = \{(a_q, b_q), (a_q s_q^2, b_q s_q^3)\}. \tag{11.4}$$

After this construction it is not known which of the curves is $E_{1,q}$ and which is $E_{2,q}$. However, by choosing points on each curve and testing whether their order is a divisor of $p^m + 1 + t$ or $p^m + 1 - t$, the curves $E_{1,q}$ and $E_{2,q}$ can be identified.

Input of our algorithm `oefCurve`(n, m, h_0) is a positive integer n (e.g. the word size of the processor in use), the prime degree m of the Optimal Extension Field over its prime field, and an integer $h_0 \geq 200$. The algorithm returns a prime p of bitlength n such that $\mathbb{F}_{p^m}$ is an Optimal Extension Field, an irreducible binomial $X^m - \omega$ in $\mathbb{F}_p[X]$, and an elliptic curve E defined over $\mathbb{F}_{p^m}$ such that $|E(\mathbb{F}_{p^m})|$ is of prime order r and respects requirements similar to those of Section 2.3.5. Furthermore, the endomorphism ring of E is a maximal order of class number at least h_0. In addition, `oefCurve` returns a generating point of $E(\mathbb{F}_{p^m})$. In order to get reasonable results we have to ensure $n \cdot m \geq 160$.

We next explain our main algorithm `oefCurve`. It splits into several subalgorithms, which we discuss in what follows. The first subalgorithm `findOEFField`(n, m, h_0) determines an Optimal Extension Field of cardinality p^m and a prime r being the group order of a cryptographically strong elliptic curve defined over $\mathbb{F}_{p^m} \setminus \mathbb{F}_p$. To be more precise, `findOEFField` computes among other things a prime p of the form $2^n + c$ with $c < 0$ and $|c| < \sqrt{2^n}$ such that $m \mid p - 1$. Although it is not clear if such a prime p exists for a random pair (n, m), the asymptotic density of such primes for growing n is $\frac{1}{(m-1)\cdot\log(2^n)}$ due to the Prime Number Theorem and a theorem of Dirichlet on the number of primes in arithmetic progressions. Hence, for example, if $n = 32$ and $m = 5$ (i.e. the case we are most interested in), there should be about $\frac{2^{16}}{4\log(2^{32})} = 739$ primes congruent 1 modulo 5 in the interval $[2^{32} - 2^{16}, 2^{32}]$. Indeed their number is 733. Thus we may assume, that an appropriate prime p exists.

In order to be successful, `findOEFField` has to solve the norm equation (11.1) for some Δ and p. We explain how to find appropriate Δ and p. A necessary condition on Δ for $|E(\mathbb{F}_{p^m})|$ to be of prime order is $\Delta \equiv 5 \bmod 8$ (see Section 4.3). Using our database fundamental_delta_h introduced on Page 75 we may assume that a sufficiently large number of fundamental imaginary quadratic discriminants $\Delta \equiv 5 \bmod 8$ of class number at least 200 is to our disposal. In our tests we make use of all such discriminants in fundamental_delta_h, that is all such fundamental discriminants with $\Delta > -6000000$. Our function `getDiscriminantOEF`(h, Δ) returns the maximal fundamental discriminant $\Delta' \equiv 5 \bmod 8$ of class number h with $\Delta' < \Delta$. In contrast to algorithm `getDiscriminant` of Chapter 5 this function only returns Δ.

In order to minimize the run time we want h and $|\Delta|$ to be as small as possible. A necessary condition, due to class field theory, we have to take care of is $m \mid h(\Delta)$. Hence we set $h = \min\{h' \in \mathbb{N} : h' \geq h_0, m \mid h'\}$. Let $\Delta \equiv 5 \bmod 8$ be maximal of class number h. We set $p = \max\{p' \in \mathbb{Z} : p' < 2^n, p' \equiv 1 \bmod m, p' \text{ prime}\}$. We determine whether the norm equation

$t^2 - \Delta y^2 = 4p$ has a solution $(t, y) \in \mathbb{Z}^2$ by using `cornacchia`(Δ, p) explained on Page 46. If $t^2 - \Delta y^2 = 4p$ has no integer solution, we turn to the norm equation $t^2 - \Delta y^2 = 4p^m$. In order to decide whether this equation has an integer solution or not, we extended the algorithm of Cornacchia to prime powers: `cornacchiaPrimePower`(Δ, p^m) gets an imaginary quadratic discriminant Δ and a prime power p^m as input. It returns $t \neq 0$ if the norm equation (11.1) has an integer solution, and 0 otherwise.

If we have found a prime p with an integer solution of the norm equation for p^m, but not for p, we make use of (11.2) to check corresponding conditions of Section 2.3.5 for prime powers. Analogously to algorithm `isStrong`(r_0, k_0, p, N), this task is performed by the function `isStrongOEF`$(2^{nm}, 1, p^m, N)$; it returns the prime r if N turns out to be the order of a cryptographically strong elliptic curve over $\mathbb{F}_{p^m}$, and 0 otherwise. This yields our algorithm `findOEFField`(n, m, h_0).

Algorithm 11.3: `findOEFField`(n, m, h_0)

Input: A positive integer n, a prime m, such that $nm \geq 160$, and an integer $h_0 \geq 200$.

Output: A prime p of bitlength n, such that $\mathbb{F}_{p^m}$ is an Optimal Extension Field, if such a p exists. A prime r and a fundamental discriminant Δ, such that r is the cardinality of a cryptographically strong elliptic curve E defined over $\mathbb{F}_{p^m} \setminus \mathbb{F}_p$ with $\mathrm{End}(E) = \mathcal{O}_\Delta$ and $h(\Delta) \geq h_0$.

```
p ← max{p' ∈ ℤ : p' < 2^n, p' ≡ 1 mod m, p' prime};
if 2^n − p > √(2^n) then
  output( "No OEF found. Terminating." ); terminate;
h ← min{h' ∈ ℕ : h' ≥ h_0, m | h'};
while true do
  Δ ← getDiscriminantOEF(h, 0);
  while Δ > −6000000 do
    p ← max{p' ∈ ℕ : p' < 2^n, p' ≡ 1 mod m, p' prime};
    while 2^n − p < √(2^n) do
      t ← cornacchia(Δ, p);
      if t = 0 then
        t ← cornacchiaPrimePower(Δ, p^m);
        if t ≠ 0 then
          if (r ← isStrongOEF(2^nm, 1, p^m, p^m + 1 − t)) ≠ 0 AND r = p^m + 1 − t then
            return(p, r, Δ);
          else if (r ← isStrongOEF(2^nm, 1, p^m, p^m + 1 + t)) ≠ 0 AND r = p^m + 1 + t then
            return(p, r, Δ);
      p ← max{p' ∈ ℤ : p' < p, p' ≡ 1 mod m, p' prime};
    Δ ← getDiscriminantOEF(h, Δ);
  h ← h + m;
```

Once knowing the cardinality p^m of an Optimal Extension Field, we turn to the computation of an irreducible binomial $X^m - \omega$ in $\mathbb{F}_p[X]$. Our algorithm `findBinomial`(p, m) is a straightforward consequence of Theorem 11.3.2 and Corollary 11.3.3.

We remark that if $X^m - \omega$ is reducible in $\mathbb{F}_p[X]$, $X^m - \omega^d$ is reducible for all $d \in \mathbb{N}$, too. However, due to the simplicity of algorithm `findBinomial`(p, m) we do not take this fact into account.

Finally, we turn to algorithm `findOEFCurve`(Δ, p^m, r). This algorithm is similar to our algo-

Algorithm 11.4: `findBinomial`(p, m)

Input: Rational primes p and m with $p \equiv 1 \bmod m$.
Output: An irreducible binomial $X^m - \omega$ in $\mathbb{F}_p[X]$ with minimal $\omega \in \mathbb{N}$.

$\omega \leftarrow 2$;
while true **do**
  $d \leftarrow \omega^{\frac{p-1}{m}} \bmod p$;
  if $d \neq 1$ **then**
    return$(X^m - \omega)$;
  $\omega \leftarrow \omega + 1$;

rithm `findCurve`. The main differences come from the subalgorithm `findOEFRoot` explained below. As said above, given a root j_q of $H \bmod p$ in $\mathbb{F}_{p^m}$, `findOEFCurve` computes the coefficients of elliptic curves over $\mathbb{F}_{p^m}$ of order $p^m + 1 \pm t$, and it decides by trial and error, which of these curves is of order r.

Algorithm 11.5: `findOEFCurve`(Δ, p^m, r)

Input: A fundamental imaginary quadratic discriminant $\Delta \equiv 5 \bmod 8$.
  A prime power p^m such that there exists an elliptic curve of prime order r over $\mathbb{F}_{p^m}$.
Output: An elliptic curve E over $\mathbb{F}_{p^m}$ with $|E(\mathbb{F}_{p^m})| = r$ and endomorphism ring of discriminant Δ.
  A generating point G of $E(\mathbb{F}_{p^m})$.

$j_q \leftarrow$ `findOEFRoot`(Δ, p^m);
Select a non-square $s_q \in \mathbb{F}_{p^m}$; $\kappa_q \leftarrow j_q/(1728 - j_q)$; $a_q \leftarrow 3\kappa_q$; $b_q \leftarrow 2\kappa_q$;
$E_1 \leftarrow (a_q, b_q)$; $E_2 \leftarrow (a_q s_q^2, b_q s_q^3)$; //assign curve parameters
$G_1 \in_R E_1(\mathbb{F}_{p^m}) \setminus \{O\}$; $G_2 \in_R E_2(\mathbb{F}_{p^m}) \setminus \{O\}$; //choose random points
if $rG_1 = O$ **then**
  return (E_1, G_1);
else
  return (E_2, G_2);

We next discuss `findOEFRoot`(Δ, p^m), i.e. the proceeding to determine a root of $H \bmod p$ in $\mathbb{F}_{p^m}$. Similar to `findRoot` the first step of `findOEFRoot`(Δ, p^m) consists in determining a generating polynomial of the ring class field of $\mathbb{Q}(\sqrt{\Delta})$. As $\Delta \equiv 5 \bmod 8$ we may not use the class polynomial W. However, if $3 \nmid \Delta$ we compute the class polynomial G according to Δ. If we have $3 \mid \Delta$, we compute the ring class polynomial H.

It remains to explain how to get a root of a class polynomial C modulo p. We extend the LiDIA-function `find_root`(p, C) to a function `find_root`(p^m, C). As input it requires a prime power p^m and a polynomial $C \in \mathbb{Z}[X]$, such that $C \bmod p$ splits into linear factors in $\mathbb{F}_{p^m}[X]$. It returns a zero of $C \bmod p$ in $\mathbb{F}_{p^m}$. Again we implement the Cantor-Zassenhaus split in `find_root`(p^m, C) and make use of a polynomial arithmetic due to Shoup [Sho95].

Algorithm 11.6: `oefCurve`(n, m, h_0)

Input: A positive integer n, a prime m, such that $nm \geq 160$, and an integer $h_0 \geq 200$.
Output: A prime p of bitlength n, such that $\mathbb{F}_{p^m}$ is an Optimal Extension Field, if such a p exists.
An irreducible binomial $X^m - \omega$ in $\mathbb{F}_p[X]$.
A cryptographically strong elliptic curve E of prime order r defined over $\mathbb{F}_{p^m} \setminus \mathbb{F}_p$ having a maximal order of class number at least h_0 as endomorphism ring.
A generating point G of $E(\mathbb{F}_{p^m})$.

$(p, r, \Delta) \leftarrow$ `findOEFField`(n, m, h_0);
$f \leftarrow$ `findBinomial`(p, m);
$(E, G) \leftarrow$ `findOEFCurve`(Δ, p^m, r);
return (p, r, f, E, G);

11.3.3 Performance and Statistics

In this section we first give sample running times measured on the Pentium III. Next, we argue that even for an input of the form $(32, 5, h_0)$, $200 \leq h_0 \leq 400$ the number of curves which may be output by our algorithm is sufficiently large. Sample curves returned by `oefCurve`$(32, 5, h_0)$ may be found in Section B.5 of the appendix.

First, we present sample output of `oefCurve`$(32, 5, h_0)$ and CPU-timings for $200 \leq h_0 \leq 250$, $5 \mid h_0$.

h_0	h	Δ	p	ω	CPU-time in seconds
200	200	-125579	4294920991	2	21.8
205	205	-140411	4294963921	2	23.5
210	210	-265235	4294903891	7	52.8
215	215	-240899	4294933021	2	43.2
220	220	-268931	4294931761	2	65.3
225	225	-316859	4294958881	2	54.4
230	230	-405803	4294931071	2	64.0
235	235	-339539	4294931341	2	53.2
240	240	-170651	4294946191	2	38.5
245	245	-411683	4294908721	5	65.0
250	250	-254579	4294940641	3	54.6

Table 11.7: Data provided by `oefCurve`$(32, 5, h_0)$.

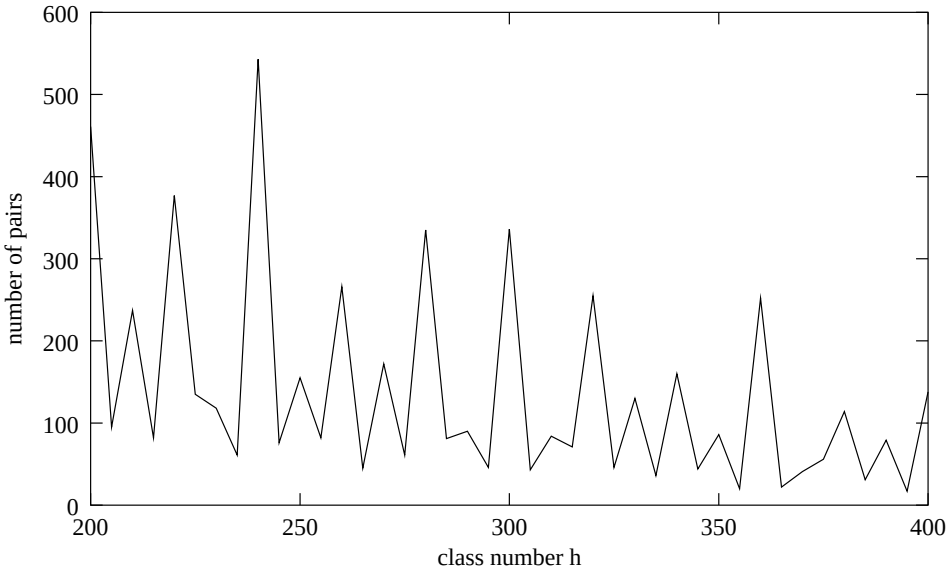
From Table 11.7 we see that in all our sample tests the output discriminant is of class number h_0. Furthermore, we deduce that the running time in practice may decrease for increasing input h_0. As such an example from Table 11.7 we consider $h_0 = 235$ and $h_0 = 240$, respectively. The reason for the decreasing running time in practice is twofold.

First, compared to $h_0 = 235$ the time of `findOEFField` is much lower in case of $h_0 = 240$. If $h_0 = 235$, we have to pass 9 times through the `while`-loop on Δ in `findOEFField`. However, if $h_0 = 240$, `findOEFField` is successful for the first chosen discriminant. Second, the floating point precision depends on both h_0 and Δ. Thus the increasing effect on the running time of an increasing h_0 may be inferior to the decreasing effect of a decreasing $|\Delta|$. Similar arguments

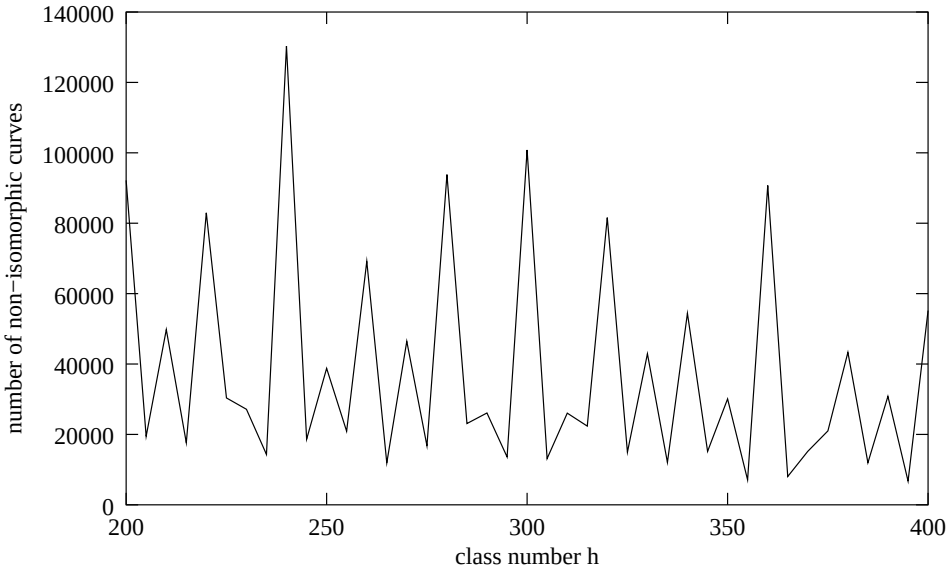
are true for other input bounds h_0, too.

Finally, we give some statistical data on the number of non-isomorphic elliptic curves of prime order over Optimal Extension Fields $\mathbb{F}_{p^5}$ where p is a 32-bit prime. Roughly speaking this quantity should be as large as possible, as in this case the output curves are not special in some sense.

First, we determine for each class number h with $200 \leq h \leq 400$, h divisible by 5, the number of pairs (Δ, p), where $\Delta > -6000000$ is a fundamental discriminant congruent 5 mod 8 and p a 32-bit prime, such that there exists a cryptographically strong elliptic curve of prime order r over $\mathbb{F}_{p^5}$ having an endomorphism ring of discriminant Δ. In all, there are 5579 such pairs. Furthermore, in 4563 of the cases, the corresponding field $\mathbb{F}_{p^5}$ is a Type II OEF. The results are plotted in the following figure.



Next, we determine the number of non-isomorphic elliptic curves for the pairs (Δ, p) as above. For each such pair (Δ, p) there are $h(\Delta)$ non-isomorphic elliptic curves having the properties cited above. In all, there are 1546830 non-isomorphic curves, and 1263850 of them are defined over a Type II OEF. The number of curves for each class number h with $200 \leq h \leq 400$ is plotted in the figure below. We deduce that, even in our special case, the set of non-isomorphic curves for use in cryptography is sufficiently large to choose from.


140000
120000
100000
80000
60000
40000
20000
0
number of non−isomorphic curves
200
250
300
350
400
class number h

Chapter 12

The Library gec

In this closing chapter we shortly present our library gec. It comprises the algorithms described in this thesis. gec allows to generate an elliptic curve suitable for use in cryptography for a chosen security level, that is for a chosen bitlength of the cryptographic relevant prime r. gec is part of the computer algebra library LiDIA ([LiDIA]). As LiDIA is implemented in C++, our library is implemented in C++, too.

In addition to the algorithms of this thesis, gec implements the random approach. The random approach makes use of the point counting package eco, which is mainly due to V. Müller ([Mül95]). eco is also part of LiDIA and implements the point counting algorithm of Schoof-Elkies-Atkin (SEA) over finite prime fields and fields of characteristic 2, respectively.

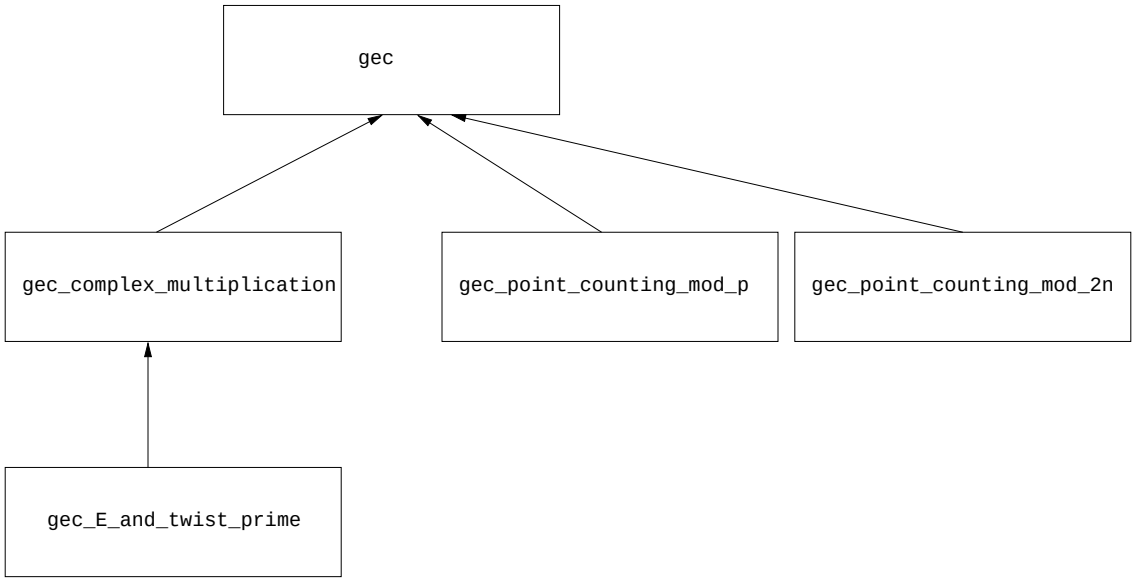


Figure 12.1: Class hierarchy of the library gec

The class hierarchy of gec is shown in Figure 12.1. The virtual basic class gec comprises the members and methods, which are common for both the CM approach and the random

approach. For instance, gec holds as members the prime field p, the prime power q, the prime r, the cofactor k, and the parameters of the elliptic curve. In addition, it stores the variables r_0, k_0, and h_0. Furthermore, it provides the accessors for these members. For more details of the members we refer to the source code of gec and to Section 12.1.

The class gec_complex_multiplication provides the algorithms described in this thesis. Furthermore, the classes gec_point_counting_mod_p and gec_point_counting_mod_2n implement the SEA-algorithm for fields $\mathbb{F}_p$ and $\mathbb{F}_{2^n}$, respectively. All these classes inherit from the base class gec. In addition, the class gec_E_and_twist_prime generates twisted pairs as explained in Section 11.2. We refer to Section 12.1 for a short description of these classes.

In Section 12.2 we describe the interface of our library gec to Java. The interface permits to generate elliptic curves suitable for use in cryptography from within the Java Cryptography Architecture (JCA, see e.g. [Knu98]). Indeed, this interface is used by the FlexiECProvider for the JCA; this provider is part of the FlexiProvider.

12.1 The Classes of gec

In this section we describe the main features of the classes of the library gec. In addition, we show a sample program to generate an elliptic curve suitable for use in cryptography.

We first turn to the base class gec. gec holds all information and provides all interfaces which are independent of the generating algorithm. However, gec does not yield a method for generating elliptic curve parameters. In Table 12.1 we list the main members of the class gec.

name in gec	name in cryptoCurve	LiDIA-type
p	p	bigint
q	q	bigint
lower_bound_bitlength_r	$\lfloor \log_2 r_0 \rfloor$	lidia_size_t
r	r	bigint
upper_bound_k	k_0	bigint
k	k	bigint
a_4	a	gf_element
a_6	b	gf_element
delta	Δ	bigint
h	$h(\Delta)$	lidia_size_t
delta_field	Δ_K	bigint
h_field	$h(\Delta_K)$	lidia_size_t
according_to_bsi	-	bool

Table 12.1: The main members of the class gec and their type.

The boolean according_to_bsi essentially decides whether the class number is taken into account as a security requirement. Furthermore, gec provides the accessors and, if available, the mutators for these members. As usual accessors and mutators may be recognized by their prefix "get" and "set", respectively. For instance, to set the prime p, one simply has to invoke the mutator set_p(const bigint &); the accessor get_p() returns the prime p. All publicly

accessible member functions are explained in the gec-manual. The complex multiplication part of the manual may be found in Chapter C of the appendix.

We next describe the class `gec_complex_multiplication`. Its most important additional member is class_polynomial of type `polynomial<bigint>`. It may be set for example by the method set_class_polynomial(`const polynomial < bigint >` &). The function generate() implements our main algorithm `cryptoCurve`.

The program in Figure 12.2 shows the easy use of our library. The program generates an elliptic curve of order rk, where r is at least of bitlength 160 and $k \leq 4$. The discriminant Δ is -21311; it is maximal and fundamental of class number 200. Hence, as $\Delta \equiv 1 \bmod 8$, we will have $k = 4$. The `boolean` bsi decides that we respect the strong requirements of GISA explained in Section 2.3.5.

```
#include <LiDIA/bigint.h>
#include <LiDIA/gec/gec_complex_multiplication.h>

int
main()
{
   lidia_size_t bitlength_r = 160;
   bigint bound_k = 4;
   bool bsi = true;
   bigint delta( -21311 );

   gec_complex_multiplication I;
   I.set_lower_bound_bitlength_r( bitlength_r );
   I.set_upper_bound_k( bound_k );
   I.set_according_to_BSI( bsi ); // respect requirements of GISA
   I.set_delta(delta);
   I.VERBOSE = 1; // print a lot of information to standard output

   I.generate(); // generate the curve parameters

   return( 0 );
}
```

Figure 12.2: A sample program using `gec_complex_multiplication`.

The use of the further classes is similar. We refer to the manual for details.

12.2 The Interface of gec to Java

We shortly describe the Java interface of our library gec. This interface enables the use of gec by classes of the Java Cryptography Architecture. We provide Java-wrapper classes, which mirror the members and functions of the classes of gec. In all, we implement the Java classes `GEC`, `GECComplexMultiplication`, and `GECPointCounting`. Currently, `GECPointCounting` only supports the random approach modulo p.

The interface makes use of the Java Native Interface (JNI). The dynamic library providing the interface functions is called jni_gec. Thus on UNIX machines the full library name is libjni_gec.so . It resides in the library directory of LiDIA.

We remark that besides the use of the interface within the JCA we have implemented two applications of the Java-wrapper classes. First, a Graphical User Interface (GUI). The GUI uses the Swing package of Java. Using the GUI one may choose between the CM-method and the random approach. In addition, one may set the security level according to one's needs. Second, we have implemented a servlet based webinterface. We refer to the website of our institute for a test of it in practice.

Appendix A

Additional Data and Information

A.1 Congruence Conditions on t and y

We supply tables to deduce the congruence relations of Section 4.3. We use the notation of Section 4.3, that is for given Δ, t, and y we set $p = (t^2 - \Delta y^2)/4$, $N_- = p + 1 - t$, and $N_- = p + 1 + t$. Furthermore, let $m \in \{3, 5, 7\}$. For all pairs $(t \bmod m, y \bmod m) \in \mathbb{F}_m^2$ we compute $p \bmod m$, $N_- \bmod m$, and $N_+ \bmod m$, respectively. Thus a necessary condition for p to be a "large" prime is $p \not\equiv 0 \bmod m$.

In addition, our aim is to have either N_- or N_+ to be nearly prime, that is of order r, $2r$ or $4r$, where r denotes a prime. Hence if both $p \not\equiv 0 \bmod m$ and $N_- \not\equiv 0 \bmod m$ for a pair $(t \bmod m, y \bmod m)$, we indicate this by plotting a star in the corresponding row. We proceed analogously with N_+.

We distinguish the two cases $\Delta \equiv 1 \bmod 8$ and $\Delta \not\equiv 1 \bmod 8$. The reason is that in the first case we know from Section 4.3.1 that both t and y are even. Hence we derive conditions for $t_{1/2}$ and $y_{1/2}$, where $t = 2t_{1/2}$ and $y = 2y_{1/2}$. However, if $\Delta \not\equiv 1 \bmod 8$ we have to consider t and y.

A.1.1 Discriminants $\Delta \equiv 1 \bmod 8$

Congruence Relations for $m = 3$

$\Delta \equiv 0 \bmod 3$						
$t_{1/2} \bmod 3$	$y_{1/2} \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	0	1		*
2	0	1	1	0	*	
0	1	0	1	1		
1	1	1	0	1		*
2	1	1	1	0	*	
0	2	0	1	1		
1	2	1	0	1		*
2	2	1	1	0	*	

$\Delta \equiv 1 \bmod 3$						
$t_{1/2} \bmod 3$	$y_{1/2} \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	0	1		*
2	0	1	1	0	*	
0	1	2	0	0		
1	1	0	2	0		
2	1	0	0	2		
0	2	2	0	0		
1	2	0	2	0		
2	2	0	0	2		

$\Delta \equiv 2 \bmod 3$						
$t_{1/2} \bmod 3$	$y_{1/2} \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	0	1		*
2	0	1	1	0	*	
0	1	1	2	2	*	*
1	1	2	1	2	*	*
2	1	2	2	1	*	*
0	2	1	2	2	*	*
1	2	2	1	2	*	*
2	2	2	2	1	*	*

Congruence Relations for $m = 5$

$\Delta \equiv 0 \bmod 5$						
$t_{1/2} \bmod 5$	$y_{1/2} \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	4	*	*
3	0	4	4	1	*	*
4	0	1	4	0	*	
0	1	0	1	1		
1	1	1	0	4		*
2	1	4	1	4	*	*
3	1	4	4	1	*	*
4	1	1	4	0	*	
0	2	0	1	1		
1	2	1	0	4		*
2	2	4	1	4	*	*
3	2	4	4	1	*	*
4	2	1	4	0	*	
0	3	0	1	1		
1	3	1	0	4		*
2	3	4	1	4	*	*
3	3	4	4	1	*	*
4	3	1	4	0	*	
0	4	0	1	1		
1	4	1	0	4		*
2	4	4	1	4	*	*
3	4	4	4	1	*	*
4	4	1	4	0	*	

$\Delta \equiv 1 \bmod 5$						
$t_{1/2} \bmod 5$	$y_{1/2} \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	4	*	*
3	0	4	4	1	*	*
4	0	1	4	0	*	
0	1	4	0	0		
1	1	0	4	3		
2	1	3	0	3		*
3	1	3	3	0	*	
4	1	0	3	4		
0	2	1	2	2	*	*
1	2	2	1	0	*	
2	2	0	2	0		
3	2	0	0	2		
4	2	2	0	1		*
0	3	1	2	2	*	*
1	3	2	1	0	*	
2	3	0	2	0		
3	3	0	0	2		
4	3	2	0	1		*
0	4	4	0	0		
1	4	0	4	3		
2	4	3	0	3		*
3	4	3	3	0	*	
4	4	0	3	4		

$\Delta \equiv 2 \bmod 5$						
$t_{1/2} \bmod 5$	$y_{1/2} \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	4	*	*
3	0	4	4	1	*	*
4	0	1	4	0	*	
0	1	3	4	4	*	*
1	1	4	3	2	*	*
2	1	2	4	2	*	*
3	1	2	2	4	*	*
4	1	4	2	3	*	*
0	2	2	3	3	*	*
1	2	3	2	1	*	*
2	2	1	3	1	*	*
3	2	1	1	3	*	*
4	2	3	1	2	*	*
0	3	2	3	3	*	*
1	3	3	2	1	*	*
2	3	1	3	1	*	*
3	3	1	1	3	*	*
4	3	3	1	2	*	*
0	4	3	4	4	*	*
1	4	4	3	2	*	*
2	4	2	4	2	*	*
3	4	2	2	4	*	*
4	4	4	2	3	*	*

$\Delta \equiv 3 \bmod 5$						
$t_{1/2} \bmod 5$	$y_{1/2} \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	4	*	*
3	0	4	4	1	*	*
4	0	1	4	0	*	
0	1	2	3	3	*	*
1	1	3	2	1	*	*
2	1	1	3	1	*	*
3	1	1	1	3	*	*
4	1	3	1	2	*	*
0	2	3	4	4	*	*
1	2	4	3	2	*	*
2	2	2	4	2	*	*
3	2	2	2	4	*	*
4	2	4	2	3	*	*
0	3	3	4	4	*	*
1	3	4	3	2	*	*
2	3	2	4	2	*	*
3	3	2	2	4	*	*
4	3	4	2	3	*	*
0	4	2	3	3	*	*
1	4	3	2	1	*	*
2	4	1	3	1	*	*
3	4	1	1	3	*	*
4	4	3	1	2	*	*

$\Delta \equiv 4 \bmod 5$						
$t_{1/2} \bmod 5$	$y_{1/2} \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	4	*	*
3	0	4	4	1	*	*
4	0	1	4	0	*	
0	1	1	2	2	*	*
1	1	2	1	0	*	
2	1	0	2	0		
3	1	0	0	2		
4	1	2	0	1		*
0	2	4	0	0		
1	2	0	4	3		
2	2	3	0	3		*
3	2	3	3	0	*	
4	2	0	3	4		
0	3	4	0	0		
1	3	0	4	3		
2	3	3	0	3		*
3	3	3	3	0	*	
4	3	0	3	4		
0	4	1	2	2	*	*
1	4	2	1	0	*	
2	4	0	2	0		
3	4	0	0	2		
4	4	2	0	1		*

Congruence Relations for $m = 7$

$\Delta \equiv 0 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	0	1	1		
1	1	1	0	4		*
2	1	4	1	2	*	*
3	1	2	4	2	*	*
4	1	2	2	4	*	*
5	1	4	2	1	*	*
6	1	1	4	0	*	
0	2	0	1	1		
1	2	1	0	4		*
2	2	4	1	2	*	*
3	2	2	4	2	*	*
4	2	2	2	4	*	*
5	2	4	2	1	*	*
6	2	1	4	0	*	
0	3	0	1	1		
1	3	1	0	4		*
2	3	4	1	2	*	*
3	3	2	4	2	*	*
4	3	2	2	4	*	*
5	3	4	2	1	*	*
6	3	1	4	0	*	
0	4	0	1	1		
1	4	1	0	4		*
2	4	4	1	2	*	*
3	4	2	4	2	*	*
4	4	2	2	4	*	*
5	4	4	2	1	*	*
6	4	1	4	0	*	
0	5	0	1	1		
1	5	1	0	4		*
2	5	4	1	2	*	*
3	5	2	4	2	*	*
4	5	2	2	4	*	*
5	5	4	2	1	*	*
6	5	1	4	0	*	
0	6	0	1	1		
1	6	1	0	4		*
2	6	4	1	2	*	*
3	6	2	4	2	*	*
4	6	2	2	4	*	*
5	6	4	2	1	*	*
6	6	1	4	0	*	

$\Delta \equiv 1 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	6	0	0		
1	1	0	6	3		
2	1	3	0	1		*
3	1	1	3	1	*	*
4	1	1	1	3	*	*
5	1	3	1	0	*	
6	1	0	3	6		
0	2	3	4	4	*	*
1	2	4	3	0	*	
2	2	0	4	5		
3	2	5	0	5		*
4	2	5	5	0	*	
5	2	0	5	4		
6	2	4	0	3		*
0	3	5	6	6	*	*
1	3	6	5	2	*	*
2	3	2	6	0	*	
3	3	0	2	0		
4	3	0	0	2		
5	3	2	0	6		*
6	3	6	2	5	*	*
0	4	5	6	6	*	*
1	4	6	5	2	*	*
2	4	2	6	0	*	
3	4	0	2	0		
4	4	0	0	2		
5	4	2	0	6		*
6	4	6	2	5	*	*
0	5	3	4	4	*	*
1	5	4	3	0	*	
2	5	0	4	5		
3	5	5	0	5		*
4	5	5	5	0	*	
5	5	0	5	4		
6	5	4	0	3		*
0	6	6	0	0		
1	6	0	6	3		
2	6	3	0	1		*
3	6	1	3	1	*	*
4	6	1	1	3	*	*
5	6	3	1	0	*	
6	6	0	3	6		

$\Delta \equiv 2 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	5	6	6	*	*
1	1	6	5	2	*	*
2	1	2	6	0	*	
3	1	0	2	0		
4	1	0	0	2		
5	1	2	0	6		*
6	1	6	2	5	*	*
0	2	6	0	0		
1	2	0	6	3		
2	2	3	0	1		*
3	2	1	3	1	*	*
4	2	1	1	3	*	*
5	2	3	1	0	*	
6	2	0	3	6		
0	3	3	4	4	*	*
1	3	4	3	0	*	
2	3	0	4	5		
3	3	5	0	5		*
4	3	5	5	0	*	
5	3	0	5	4		
6	3	4	0	3		*
0	4	3	4	4	*	*
1	4	4	3	0	*	
2	4	0	4	5		
3	4	5	0	5		*
4	4	5	5	0	*	
5	4	0	5	4		
6	4	4	0	3		*
0	5	6	0	0		
1	5	0	6	3		
2	5	3	0	1		*
3	5	1	3	1	*	*
4	5	1	1	3	*	*
5	5	3	1	0	*	
6	5	0	3	6		
0	6	5	6	6	*	*
1	6	6	5	2	*	*
2	6	2	6	0	*	
3	6	0	2	0		
4	6	0	0	2		
5	6	2	0	6		*
6	6	6	2	5	*	*

$\Delta \equiv 3 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	4	5	5	*	*
1	1	5	4	1	*	*
2	1	1	5	6	*	*
3	1	6	1	6	*	*
4	1	6	6	1	*	*
5	1	1	6	5	*	*
6	1	5	1	4	*	*
0	2	2	3	3	*	*
1	2	3	2	6	*	*
2	2	6	3	4	*	*
3	2	4	6	4	*	*
4	2	4	4	6	*	*
5	2	6	4	3	*	*
6	2	3	6	2	*	*
0	3	1	2	2	*	*
1	3	2	1	5	*	*
2	3	5	2	3	*	*
3	3	3	5	3	*	*
4	3	3	3	5	*	*
5	3	5	3	2	*	*
6	3	2	5	1	*	*
0	4	1	2	2	*	*
1	4	2	1	5	*	*
2	4	5	2	3	*	*
3	4	3	5	3	*	*
4	4	3	3	5	*	*
5	4	5	3	2	*	*
6	4	2	5	1	*	*
0	5	2	3	3	*	*
1	5	3	2	6	*	*
2	5	6	3	4	*	*
3	5	4	6	4	*	*
4	5	4	4	6	*	*
5	5	6	4	3	*	*
6	5	3	6	2	*	*
0	6	4	5	5	*	*
1	6	5	4	1	*	*
2	6	1	5	6	*	*
3	6	6	1	6	*	*
4	6	6	6	1	*	*
5	6	1	6	5	*	*
6	6	5	1	4	*	*

$\Delta \equiv 4 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	3	4	4	*	*
1	1	4	3	0	*	
2	1	0	4	5		
3	1	5	0	5		*
4	1	5	5	0	*	
5	1	0	5	4		
6	1	4	0	3		*
0	2	5	6	6	*	*
1	2	6	5	2	*	*
2	2	2	6	0	*	
3	2	0	2	0		
4	2	0	0	2		
5	2	2	0	6		*
6	2	6	2	5	*	*
0	3	6	0	0		
1	3	0	6	3		
2	3	3	0	1		*
3	3	1	3	1	*	*
4	3	1	1	3	*	*
5	3	3	1	0	*	
6	3	0	3	6		
0	4	6	0	0		
1	4	0	6	3		
2	4	3	0	1		*
3	4	1	3	1	*	*
4	4	1	1	3	*	*
5	4	3	1	0	*	
6	4	0	3	6		
0	5	5	6	6	*	*
1	5	6	5	2	*	*
2	5	2	6	0	*	
3	5	0	2	0		
4	5	0	0	2		
5	5	2	0	6		*
6	5	6	2	5	*	*
0	6	3	4	4	*	*
1	6	4	3	0	*	
2	6	0	4	5		
3	6	5	0	5		*
4	6	5	5	0	*	
5	6	0	5	4		
6	6	4	0	3		*

$\Delta \equiv 5 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	2	3	3	*	*
1	1	3	2	6	*	*
2	1	6	3	4	*	*
3	1	4	6	4	*	*
4	1	4	4	6	*	*
5	1	6	4	3	*	*
6	1	3	6	2	*	*
0	2	1	2	2	*	*
1	2	2	1	5	*	*
2	2	5	2	3	*	*
3	2	3	5	3	*	*
4	2	3	3	5	*	*
5	2	5	3	2	*	*
6	2	2	5	1	*	*
0	3	4	5	5	*	*
1	3	5	4	1	*	*
2	3	1	5	6	*	*
3	3	6	1	6	*	*
4	3	6	6	1	*	*
5	3	1	6	5	*	*
6	3	5	1	4	*	*
0	4	4	5	5	*	*
1	4	5	4	1	*	*
2	4	1	5	6	*	*
3	4	6	1	6	*	*
4	4	6	6	1	*	*
5	4	1	6	5	*	*
6	4	5	1	4	*	*
0	5	1	2	2	*	*
1	5	2	1	5	*	*
2	5	5	2	3	*	*
3	5	3	5	3	*	*
4	5	3	3	5	*	*
5	5	5	3	2	*	*
6	5	2	5	1	*	*
0	6	2	3	3	*	*
1	6	3	2	6	*	*
2	6	6	3	4	*	*
3	6	4	6	4	*	*
4	6	4	4	6	*	*
5	6	6	4	3	*	*
6	6	3	6	2	*	*

$\Delta \equiv 6 \bmod 7$						
$t_{1/2} \bmod 7$	$y_{1/2} \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	1	0	4		*
2	0	4	1	2	*	*
3	0	2	4	2	*	*
4	0	2	2	4	*	*
5	0	4	2	1	*	*
6	0	1	4	0	*	
0	1	1	2	2	*	*
1	1	2	1	5	*	*
2	1	5	2	3	*	*
3	1	3	5	3	*	*
4	1	3	3	5	*	*
5	1	5	3	2	*	*
6	1	2	5	1	*	*
0	2	4	5	5	*	*
1	2	5	4	1	*	*
2	2	1	5	6	*	*
3	2	6	1	6	*	*
4	2	6	6	1	*	*
5	2	1	6	5	*	*
6	2	5	1	4	*	*
0	3	2	3	3	*	*
1	3	3	2	6	*	*
2	3	6	3	4	*	*
3	3	4	6	4	*	*
4	3	4	4	6	*	*
5	3	6	4	3	*	*
6	3	3	6	2	*	*
0	4	2	3	3	*	*
1	4	3	2	6	*	*
2	4	6	3	4	*	*
3	4	4	6	4	*	*
4	4	4	4	6	*	*
5	4	6	4	3	*	*
6	4	3	6	2	*	*
0	5	4	5	5	*	*
1	5	5	4	1	*	*
2	5	1	5	6	*	*
3	5	6	1	6	*	*
4	5	6	6	1	*	*
5	5	1	6	5	*	*
6	5	5	1	4	*	*
0	6	1	2	2	*	*
1	6	2	1	5	*	*
2	6	5	2	3	*	*
3	6	3	5	3	*	*
4	6	3	3	5	*	*
5	6	5	3	2	*	*
6	6	2	5	1	*	*

A.1.2 Discriminants $\Delta \not\equiv 1 \bmod 8$

Congruence Relations for $m = 3$

$\Delta \equiv 0 \bmod 3$						
$t \bmod 3$	$y \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	1	0	*	
2	0	1	0	1		*
0	1	0	1	1		
1	1	1	1	0	*	
2	1	1	0	1		*
0	2	0	1	1		
1	2	1	1	0	*	
2	2	1	0	1		*

$\Delta \equiv 1 \bmod 3$						
$t \bmod 3$	$y \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	1	0	*	
2	0	1	0	1		*
0	1	2	0	0		
1	1	0	0	2		
2	1	0	2	0		
0	2	2	0	0		
1	2	0	0	2		
2	2	0	2	0		

$\Delta \equiv 2 \bmod 3$						
$t \bmod 3$	$y \bmod 3$	$p \bmod 3$	$N_- \bmod 3$	$N_+ \bmod 3$	N_-	N_+
0	0	0	1	1		
1	0	1	1	0	*	
2	0	1	0	1		*
0	1	1	2	2	*	*
1	1	2	2	1	*	*
2	1	2	1	2	*	*
0	2	1	2	2	*	*
1	2	2	2	1	*	*
2	2	2	1	2	*	*

Congruence Relations for $m = 5$

$\Delta \equiv 0 \bmod 5$						
$t \bmod 5$	$y \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	4	4	1	*	*
2	0	1	0	4		*
3	0	1	4	0	*	
4	0	4	1	4	*	*
0	1	0	1	1		
1	1	4	4	1	*	*
2	1	1	0	4		*
3	1	1	4	0	*	
4	1	4	1	4	*	*
0	2	0	1	1		
1	2	4	4	1	*	*
2	2	1	0	4		*
3	2	1	4	0	*	
4	2	4	1	4	*	*
0	3	0	1	1		
1	3	4	4	1	*	*
2	3	1	0	4		*
3	3	1	4	0	*	
4	3	4	1	4	*	*
0	4	0	1	1		
1	4	4	4	1	*	*
2	4	1	0	4		*
3	4	1	4	0	*	
4	4	4	1	4	*	*

$\Delta \equiv 1 \bmod 5$						
$t \bmod 5$	$y \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	4	4	1	*	*
2	0	1	0	4		*
3	0	1	4	0	*	
4	0	4	1	4	*	*
0	1	1	2	2	*	*
1	1	0	0	2		
2	1	2	1	0	*	
3	1	2	0	1		*
4	1	0	2	0		
0	2	4	0	0		
1	2	3	3	0	*	
2	2	0	4	3		
3	2	0	3	4		
4	2	3	0	3		*
0	3	4	0	0		
1	3	3	3	0	*	
2	3	0	4	3		
3	3	0	3	4		
4	3	3	0	3		*
0	4	1	2	2	*	*
1	4	0	0	2		
2	4	2	1	0	*	
3	4	2	0	1		*
4	4	0	2	0		

$\Delta \equiv 2 \bmod 5$						
$t \bmod 5$	$y \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	4	4	1	*	*
2	0	1	0	4		*
3	0	1	4	0	*	
4	0	4	1	4	*	*
0	1	2	3	3	*	*
1	1	1	1	3	*	*
2	1	3	2	1	*	*
3	1	3	1	2	*	*
4	1	1	3	1	*	*
0	2	3	4	4	*	*
1	2	2	2	4	*	*
2	2	4	3	2	*	*
3	2	4	2	3	*	*
4	2	2	4	2	*	*
0	3	3	4	4	*	*
1	3	2	2	4	*	*
2	3	4	3	2	*	*
3	3	4	2	3	*	*
4	3	2	4	2	*	*
0	4	2	3	3	*	*
1	4	1	1	3	*	*
2	4	3	2	1	*	*
3	4	3	1	2	*	*
4	4	1	3	1	*	*

$\Delta \equiv 3 \bmod 5$						
$t \bmod 5$	$y \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	4	4	1	*	*
2	0	1	0	4		*
3	0	1	4	0	*	
4	0	4	1	4	*	*
0	1	3	4	4	*	*
1	1	2	2	4	*	*
2	1	4	3	2	*	*
3	1	4	2	3	*	*
4	1	2	4	2	*	*
0	2	2	3	3	*	*
1	2	1	1	3	*	*
2	2	3	2	1	*	*
3	2	3	1	2	*	*
4	2	1	3	1	*	*
0	3	2	3	3	*	*
1	3	1	1	3	*	*
2	3	3	2	1	*	*
3	3	3	1	2	*	*
4	3	1	3	1	*	*
0	4	3	4	4	*	*
1	4	2	2	4	*	*
2	4	4	3	2	*	*
3	4	4	2	3	*	*
4	4	2	4	2	*	*

$\Delta \equiv 4 \bmod 5$						
$t \bmod 5$	$y \bmod 5$	$p \bmod 5$	$N_- \bmod 5$	$N_+ \bmod 5$	N_-	N_+
0	0	0	1	1		
1	0	4	4	1	*	*
2	0	1	0	4		*
3	0	1	4	0	*	
4	0	4	1	4	*	*
0	1	4	0	0		
1	1	3	3	0	*	
2	1	0	4	3		
3	1	0	3	4		
4	1	3	0	3		*
0	2	1	2	2	*	*
1	2	0	0	2		
2	2	2	1	0	*	
3	2	2	0	1		*
4	2	0	2	0		
0	3	1	2	2	*	*
1	3	0	0	2		
2	3	2	1	0	*	
3	3	2	0	1		*
4	3	0	2	0		
0	4	4	0	0		
1	4	3	3	0	*	
2	4	0	4	3		
3	4	0	3	4		
4	4	3	0	3		*

Congruence Relations for $m = 7$

$\Delta \equiv 0 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	0	1	1		
1	1	2	2	4	*	*
2	1	1	0	4		*
3	1	4	2	1	*	*
4	1	4	1	2	*	*
5	1	1	4	0	*	
6	1	2	4	2	*	*
0	2	0	1	1		
1	2	2	2	4	*	*
2	2	1	0	4		*
3	2	4	2	1	*	*
4	2	4	1	2	*	*
5	2	1	4	0	*	
6	2	2	4	2	*	*
0	3	0	1	1		
1	3	2	2	4	*	*
2	3	1	0	4		*
3	3	4	2	1	*	*
4	3	4	1	2	*	*
5	3	1	4	0	*	
6	3	2	4	2	*	*
0	4	0	1	1		
1	4	2	2	4	*	*
2	4	1	0	4		*
3	4	4	2	1	*	*
4	4	4	1	2	*	*
5	4	1	4	0	*	
6	4	2	4	2	*	*
0	5	0	1	1		
1	5	2	2	4	*	*
2	5	1	0	4		*
3	5	4	2	1	*	*
4	5	4	1	2	*	*
5	5	1	4	0	*	
6	5	2	4	2	*	*
0	6	0	1	1		
1	6	2	2	4	*	*
2	6	1	0	4		*
3	6	4	2	1	*	*
4	6	4	1	2	*	*
5	6	1	4	0	*	
6	6	2	4	2	*	*

$\Delta \equiv 1 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	5	6	6	*	*
1	1	0	0	2		
2	1	6	5	2	*	*
3	1	2	0	6		*
4	1	2	6	0	*	
5	1	6	2	5	*	*
6	1	0	2	0		
0	2	6	0	0		
1	2	1	1	3	*	*
2	2	0	6	3		
3	2	3	1	0	*	
4	2	3	0	1		*
5	2	0	3	6		
6	2	1	3	1	*	*
0	3	3	4	4	*	*
1	3	5	5	0	*	
2	3	4	3	0	*	
3	3	0	5	4		
4	3	0	4	5		
5	3	4	0	3		*
6	3	5	0	5		*
0	4	3	4	4	*	*
1	4	5	5	0	*	
2	4	4	3	0	*	
3	4	0	5	4		
4	4	0	4	5		
5	4	4	0	3		*
6	4	5	0	5		*
0	5	6	0	0		
1	5	1	1	3	*	*
2	5	0	6	3		
3	5	3	1	0	*	
4	5	3	0	1		*
5	5	0	3	6		
6	5	1	3	1	*	*
0	6	5	6	6	*	*
1	6	0	0	2		
2	6	6	5	2	*	*
3	6	2	0	6		*
4	6	2	6	0	*	
5	6	6	2	5	*	*
6	6	0	2	0		

$\Delta \equiv 2 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	3	4	4	*	*
1	1	5	5	0	*	
2	1	4	3	0	*	
3	1	0	5	4		
4	1	0	4	5		
5	1	4	0	3		*
6	1	5	0	5		*
0	2	5	6	6	*	*
1	2	0	0	2		
2	2	6	5	2	*	*
3	2	2	0	6		*
4	2	2	6	0	*	
5	2	6	2	5	*	*
6	2	0	2	0		
0	3	6	0	0		
1	3	1	1	3	*	*
2	3	0	6	3		
3	3	3	1	0	*	
4	3	3	0	1		*
5	3	0	3	6		
6	3	1	3	1	*	*
0	4	6	0	0		
1	4	1	1	3	*	*
2	4	0	6	3		
3	4	3	1	0	*	
4	4	3	0	1		*
5	4	0	3	6		
6	4	1	3	1	*	*
0	5	5	6	6	*	*
1	5	0	0	2		
2	5	6	5	2	*	*
3	5	2	0	6		*
4	5	2	6	0	*	
5	5	6	2	5	*	*
6	5	0	2	0		
0	6	3	4	4	*	*
1	6	5	5	0	*	
2	6	4	3	0	*	
3	6	0	5	4		
4	6	0	4	5		
5	6	4	0	3		*
6	6	5	0	5		*

$\Delta \equiv 3 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	1	2	2	*	*
1	1	3	3	5	*	*
2	1	2	1	5	*	*
3	1	5	3	2	*	*
4	1	5	2	3	*	*
5	1	2	5	1	*	*
6	1	3	5	3	*	*
0	2	4	5	5	*	*
1	2	6	6	1	*	*
2	2	5	4	1	*	*
3	2	1	6	5	*	*
4	2	1	5	6	*	*
5	2	5	1	4	*	*
6	2	6	1	6	*	*
0	3	2	3	3	*	*
1	3	4	4	6	*	*
2	3	3	2	6	*	*
3	3	6	4	3	*	*
4	3	6	3	4	*	*
5	3	3	6	2	*	*
6	3	4	6	4	*	*
0	4	2	3	3	*	*
1	4	4	4	6	*	*
2	4	3	2	6	*	*
3	4	6	4	3	*	*
4	4	6	3	4	*	*
5	4	3	6	2	*	*
6	4	4	6	4	*	*
0	5	4	5	5	*	*
1	5	6	6	1	*	*
2	5	5	4	1	*	*
3	5	1	6	5	*	*
4	5	1	5	6	*	*
5	5	5	1	4	*	*
6	5	6	1	6	*	*
0	6	1	2	2	*	*
1	6	3	3	5	*	*
2	6	2	1	5	*	*
3	6	5	3	2	*	*
4	6	5	2	3	*	*
5	6	2	5	1	*	*
6	6	3	5	3	*	*

$\Delta \equiv 4 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	6	0	0		
1	1	1	1	3	*	*
2	1	0	6	3		
3	1	3	1	0	*	
4	1	3	0	1		*
5	1	0	3	6		
6	1	1	3	1	*	*
0	2	3	4	4	*	*
1	2	5	5	0	*	
2	2	4	3	0	*	
3	2	0	5	4		
4	2	0	4	5		
5	2	4	0	3		*
6	2	5	0	5		*
0	3	5	6	6	*	*
1	3	0	0	2		
2	3	6	5	2	*	*
3	3	2	0	6		*
4	3	2	6	0	*	
5	3	6	2	5	*	*
6	3	0	2	0		
0	4	5	6	6	*	*
1	4	0	0	2		
2	4	6	5	2	*	*
3	4	2	0	6		*
4	4	2	6	0	*	
5	4	6	2	5	*	*
6	4	0	2	0		
0	5	3	4	4	*	*
1	5	5	5	0	*	
2	5	4	3	0	*	
3	5	0	5	4		
4	5	0	4	5		
5	5	4	0	3		*
6	5	5	0	5		*
0	6	6	0	0		
1	6	1	1	3	*	*
2	6	0	6	3		
3	6	3	1	0	*	
4	6	3	0	1		*
5	6	0	3	6		
6	6	1	3	1	*	*

$\Delta \equiv 5 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	4	5	5	*	*
1	1	6	6	1	*	*
2	1	5	4	1	*	*
3	1	1	6	5	*	*
4	1	1	5	6	*	*
5	1	5	1	4	*	*
6	1	6	1	6	*	*
0	2	2	3	3	*	*
1	2	4	4	6	*	*
2	2	3	2	6	*	*
3	2	6	4	3	*	*
4	2	6	3	4	*	*
5	2	3	6	2	*	*
6	2	4	6	4	*	*
0	3	1	2	2	*	*
1	3	3	3	5	*	*
2	3	2	1	5	*	*
3	3	5	3	2	*	*
4	3	5	2	3	*	*
5	3	2	5	1	*	*
6	3	3	5	3	*	*
0	4	1	2	2	*	*
1	4	3	3	5	*	*
2	4	2	1	5	*	*
3	4	5	3	2	*	*
4	4	5	2	3	*	*
5	4	2	5	1	*	*
6	4	3	5	3	*	*
0	5	2	3	3	*	*
1	5	4	4	6	*	*
2	5	3	2	6	*	*
3	5	6	4	3	*	*
4	5	6	3	4	*	*
5	5	3	6	2	*	*
6	5	4	6	4	*	*
0	6	4	5	5	*	*
1	6	6	6	1	*	*
2	6	5	4	1	*	*
3	6	1	6	5	*	*
4	6	1	5	6	*	*
5	6	5	1	4	*	*
6	6	6	1	6	*	*

$\Delta \equiv 6 \bmod 7$						
$t \bmod 7$	$y \bmod 7$	$p \bmod 7$	$N_- \bmod 7$	$N_+ \bmod 7$	N_-	N_+
0	0	0	1	1		
1	0	2	2	4	*	*
2	0	1	0	4		*
3	0	4	2	1	*	*
4	0	4	1	2	*	*
5	0	1	4	0	*	
6	0	2	4	2	*	*
0	1	2	3	3	*	*
1	1	4	4	6	*	*
2	1	3	2	6	*	*
3	1	6	4	3	*	*
4	1	6	3	4	*	*
5	1	3	6	2	*	*
6	1	4	6	4	*	*
0	2	1	2	2	*	*
1	2	3	3	5	*	*
2	2	2	1	5	*	*
3	2	5	3	2	*	*
4	2	5	2	3	*	*
5	2	2	5	1	*	*
6	2	3	5	3	*	*
0	3	4	5	5	*	*
1	3	6	6	1	*	*
2	3	5	4	1	*	*
3	3	1	6	5	*	*
4	3	1	5	6	*	*
5	3	5	1	4	*	*
6	3	6	1	6	*	*
0	4	4	5	5	*	*
1	4	6	6	1	*	*
2	4	5	4	1	*	*
3	4	1	6	5	*	*
4	4	1	5	6	*	*
5	4	5	1	4	*	*
6	4	6	1	6	*	*
0	5	1	2	2	*	*
1	5	3	3	5	*	*
2	5	2	1	5	*	*
3	5	5	3	2	*	*
4	5	5	2	3	*	*
5	5	2	5	1	*	*
6	5	3	5	3	*	*
0	6	2	3	3	*	*
1	6	4	4	6	*	*
2	6	3	2	6	*	*
3	6	6	4	3	*	*
4	6	6	3	4	*	*
5	6	3	6	2	*	*
6	6	4	6	4	*	*

A.2 Sample Tests of Algorithm `findDiscriminant`

In this section we present sample output of algorithm `findDiscriminant`(p, r_0, k_0, h_0) for primes p of different bitlength. Let p be a prime of bitlength b. In our tests, we make use of $b = 160$ and $b = 250$, respectively. We tested `findDiscriminant`(p, r_0, k_0, h_0) for $r_0 = 2^{b-1}$, $k_0 = 1$, and $h_0 = 200$. For fixed b, we performed 100 tests on the Pentium III. In order to perform the tests, we first determined 100 random primes of bitlength 160 and 250, respectively.

In addition, we determined the timing of the whole generating algorithm `cryptoCurve`. All timings are given in seconds and come from the tests on the Pentium III. The rows are sorted in increasing order with respect to the timing of `cryptoCurve`.

The output for $b = 160$ is given in the tables on the Pages 196 - 198. We present the discriminant Δ, its class number $h(\Delta)$, the prime r, the cofactor $k = 1$, and the timings of `findDiscriminant` and `cryptoCurve` in seconds. Furthermore, the output for $b = 250$ is given in the tables on the Pages 199 - 201. We replace the listing of r by its bitlength $\lfloor \log_2 r \rfloor + 1$.

p	Δ	h	r	k	time of find-Discriminant	time of crypto-Curve
889345481304130030990723089105400067250279394663	-988867	200	889345481304130030990724092915125419006407092737	1	0.21	26.38
916189538556322855512698504590445225435023953679	-866131	200	916189538556322855512696594141246354798254627201	1	0.58	26.94
978065276973228012871936967012709267059489992061	-893659	200	978065276973228012871937102871291755550519692687	1	0.54	27.04
109992890844862090065087427898834606120940667453 33	-1702363	200	109992890844862090065087506487904554501283845783 49	1	0.89	31.5
142383737402091575731204391000932260504959259115 1	-949291	204	142383737402091575731204569515842987339726088677 77	1	5.35	33.17
135525575340788041873713482643073920466322720642 9	-1262059	202	135525575340788041873713360129819289396700527920 1	1	2.76	33.58
118415792217467723065760134972227310399056481923 1	-559619	204	118415792217467723065760339223385268374730529555 7	1	5.27	36.19
974179151283745932189345841534183909319990588587	-3306403	200	974179151283745932189344817979244172271023053603	1	0.75	37.02
127385417603309862843038404848127409851196873577963	-3228283	200	127385417603309862843038255189063165952816331229 77	1	1.28	37.16
107671998589394671497691836314277120891374141985 7	-1724707	202	107671998589394671497691847714447246126191999702 3	1	3.09	37.17
137934495145009496933076682129646319818720741933 9	-1672339	201	137934495145009496933076793578126542488719754031	1	3.2	37.27
142410783179925267311851257896357537429929640796 7	-2602027	200	142410783179925267311851480163904370492713112234 7	1	1.66	38.76
105075021188304282071386405147254894797000964057 07	-3544963	200	105075021188304282071386440712349313723571814605 7	1	1.76	39.25
123453279643076326241775526410152482709864549710 9	-1973899	202	123453279643076326241775566733113388611539752463 29	1	4.01	40.11
837997517206705355425727328792071381688981190907	-4075747	200	837997517206705355425728878627701904133812402993	1	1.23	41.09
102422981031977642679955254476307213745678614093 1	-1337251	204	102422981031977642679955067860610850928154051 07	1	6.27	41.54
142279363430976315440649260963603943210354327383 3	-1557691	208	142279363430976315440649457417210112503156849820 3	1	8.77	42.11
108825473828311549216618840047465344404737211171 7	-3381427	202	108825473828311549216618876101048969136104749109 7	1	3.91	42.39
120360224310225985383673901498721923207016036873	-2894827	202	120360224310225985383671727483644635714838697809	1	4.27	42.98
765503541934501829275966065218712872190774896309	-1088059	208	765503541934501829275965439551120858906384428591	1	12.25	43
144071248566674550373794643834533796630491333170 33	-1302643	210	144071248566674550373794408168333333063196088482 9	1	11.35	43.18
980237189411139094568439826587129091843711878857	-3054643	205	980237189411139094568437924453444832986140645263	1	7.41	43.52
106979173415882204830529285953850310889865118492 11	-2513467	204	106979173415882204830529309529261990553813013987 3	1	7.07	44.16
768674824563172326953826348887499474832122937967 9	-2106187	206	768674824563172326958262185979815539874081281997	1	8.48	45.15
856194244212583603450653142307658739535654308409	-2655667	204	856194244212583603450651955503930535282627596944 7	1	8.13	45.16
142919987397509939729771607271374170834170654236 1	-5170867	202	142919987397509939729771829014017959993399151349 9	1	4.06	46.38
144238076409155807862641222882061551787401224375 3	-5575003	200	144238076409155807862640993804628433493046427010 1	1	1.64	46.71
145480413826566399524278010142493436029239295015 1	-1139731	210	145480413826566399524278162915469983988896146663 7	1	13.42	46.95
129640254843883176828742898481817495209750611935 9	-1533355	208	129640254843883176828743047136382504587749690046 9	1	10.7	46.99
899051559993030661121188797869222006763186941921	-4721803	202	899051559993030661121189111677951474582776569669	1	4.92	47.15
761531905094886738984943559442966665936581089333	-1441339	211	761531905094886738984942905793298152965238204073	1	12.77	47.33
105448238210808328182148429200296687986943474476 7	-4018987	204	105448238210808328182148315537166778962329336261 3	1	5.82	47.36
104383709223358388463824516509536176476147221622 1	-2816947	204	104383709223358388463824387750719286661011336919 1	1	7.95	47.38
130591946481579802917609149022540998007543001666 7	-2308363	208	130591946481579802917608926256270441102878018266 7	1	9.29	47.51
144647559837826811197811492903510589978335785872 1	-1605667	208	144647559837826811197811714008657190711472794932 1	1	13.98	47.62
927371529347532766869974048229414040887816385249 1	-3545803	204	927371529347532766869740968396976014136571100403	1	9.73	48.22
795704590944348192856717644666562049985058379977	-3635107	206	795704590944348192856716936694029012404964726539	1	8.5	48.22
815475019600533370128883986508177117824414405917	-6057283	202	815475019600533370128883325423865002049392683709	1	3.76	48.35
859065927319643642245909496358874899291829133591	-1390315	206	859065927319643642245907695710920057960102048419	1	12.18	48.38
116195589997312982761723345951606321619827682404 1	-3984115	204	116195589997312982761723132360776506151786806011 9	1	8.38	48.66

p	Δ	h	r	k	time of findDiscriminant	time of cryptoCurve
13600931967873325614311513485289488085474082133 51	-3049435	206	136009319678733256143115314704137227717879210896 9	1	10	48.89
14019475207409455718286629750064295849485694488 77	-3439027	208	140194752074094557182866459745783747297369724957898813	1	10.51	49.04
11031537328125133878850161202555070773739317470411	-1836115	208	110315373281251338788501685323857374757238827719	1	13.33	49.32
13660117917325881401442009752770109264685474517090	-3499003	204	136601179173258814014420015875229336359883412652 9	1	9.69	49.97
85164241809157609577236850629026417827086555077 7	-5208547	204	851642418091576095772366986881947577561298341999	1	7.68	50.5
11646423379553908340615162334139092307460719276 57	-1098563	214	116464233795539083406151435846595421791878686358 7	1	14.76	51.12
10772502742563406506457276359428730711993453607 01	-1881955	208	107725027425634065064572915048770994729889263745 9	1	14.07	51.54
13717638926993320904884073600564250986335011973 67	-3595747	210	137176389269933209048840706412186372404920083342 7	1	11.82	51.57
80404652828357254852271936681283872460502883656 1	-1732795	208	8040465282835725485227188069065437533290790664 79	1	16.21	52.53
11301419721534204092113637033056931994045215114 3	-2695843	208	113014197215342040921136477632804333281637298375 7	1	14.21	53.9
12064278902183549789331122653205127075421890280 09	-3140347	208	120642789021835497893311381409001364506262790456 3	1	16.24	54.92
96763824799342243458009755877012846277269010796 9	-833395	216	9676382479934224345800956234527676197653128277270 9	1	25.07	55.73
12160371491581397116947793465067395930200221982 49	-1997947	210	121603714915813971169477844348223191429569865589	1	20.12	56.13
88384667653725531879432004201819584439337022004 3	-1143227	217	883846676537255318794319684654149777096911040009	1	17.56	56.45
11061724161003584259340397079308347484938431180 67	-1221499	217	110617241610035842559340417425556860990979497242001	1	20.04	56.82
10619209079231095189767992636801334803877318271 49	-3569995	208	106192090792310951897680132152744909992188243735 1	1	14.8	57
11947054536458379238516098889638405837962160784 01	-1164091	217	119470545364583792385161207059173046660690052119 7	1	23.1	58.99
88715915425001218392821375634512332410106373140 3	-2974723	217	8871591542500121839282120835604642389773839147 59	1	20.1	59.78
11939456477503385492961210670695060724704459184 79	-3679435	212	119394564775033854929612120646423027832837526633 9	1	17.8	60.58
73114299522166241726541225640131676282732190299 9	-4777507	208	731142995221662417265414332009692959858597231623 3	1	14.49	60.62
12139566588506339845089481358165485279674324329 77	-10184427	218	121395666588506339845089467665934515693903890655 99	1	22.89	62.11
12356974542719523990980909037872487574959840324683	-1873531	216	123569745427195239909808913116286892254382857860 3	1	21.12	62.47
83016237176808633449287536947214921944977247207	-5451667	212	830162371768086334492887410907423205414045169943 3	1	17.03	63.02
97182857422401415299725559420835716173775260154 7	-2503963	216	971828574224014152997255414534894802779811321603	1	22.2	63.24
86935790597870029791251310692757349460981992256 9	-4293667	214	8693579059787002979125111816966828168837274582921	1	18.21	63.41
12679119511635716530236079825443249550487247274 91	-2860123	212	126791195116357165302360598118505009217411329478 1	1	26.06	65.84
10551314694987057301533210264766300246572313057 37	-2870683	219	105513146949870573015332089852181048241030889286 3	1	27.09	67.83
78902602609090977234901343593929949050916679494 3	-1830211	223	789026026090909772349014548581656607001251116963	1	26.9	68.18
80275814572322899750858883034762433658314722723 7	-1225363	224	802758145723228997508590537280131007738273509939	1	32.42	68.75
14486723462876214802527248965851237137763185603 03	-3791467	222	144867234628762148025272350275663703342971919821 1	1	22.18	69.36
77695852649491510080268738739102671111708820903	-7341123	228	776958526949151008026874781989077103375287703491	1	34.42	70.29
13289678169848202407466490649670492576956461700 29	-2266003	217	1328967816984820240746649067065979615493546308626351	1	29.6	70.88
13560962137306881969258517911579687911632234858 27	-1491259	224	135609621373068819692585241295158413369818760481 1	1	31.22	71.92
11868536341156061952795992190623626184113347247 37	-4168003	216	118685363411560619527960111935078285599451320157 63	1	25.21	72.16
14475960647642421356282639442494736679142847838 9	-1798555	224	144759606476424421356282757383222021538031382730 9	1	33.35	73.88
12864334376755897923289759796988247720105958522 29	-3623995	220	128643343767558979232897512481505001646230447709 9	1	34.83	78.56
11720776107388226301158742962358093675366870729 77	-1628443	232	1172077610738822630115872447136876796516759103777	1	42.94	82.47
12995589927371221975337617214166998574394660950 23	-2736643	224	129955899273712219753376759745766534833901637131799	1	34.79	82.69
79272926853257850445912989664378332599793249329	-819083	223	7927229268532578504459118723876955997685587742 51	1	44.97	82.77
80722044847467548944227496452647144084137945513 9	-4464427	220	807220448474675489442273696651843947711081625581	1	37.5	83.52

p	Δ	h	r	k	time of find-Discriminant	time of crypto-Curve
897675185299036805830804402985599800620274411121	-589499	228	897675185299036805830804609832083096903575787297	1	47.58	84.51
1055556226996553381850435356640347702535370318387	-2584123	234	1055556226996553381850436026035205893210560810327	1	38.98	86.98
1208155041965007520265014253146636752469564538501	-1938595	228	1208155041965007520265014858560331095216757861879	1	49.84	92.67
1168914892804667346784000850717482275996373988249	-1128595	232	1168914892804667346784001284939635899166739061261	1	55.97	94.12
990397830669707771020420203992421980784833212979 7	-1946923	232	990397830669707771020420176254055868097514678038 7	1	51.68	94.82
127904414280692688459914042578192219922033916000 1	-2031019	232	127904414280692688459914117247781582492602993127 9	1	49.74	94.97
96116242723131163042680814034424479168648524852 9	-1960915	232	96116242723131163042680711066711720312874211933 9	1	51.36	95.06
13286098376944350678155493307489806462042810994 49	-3719323	225	13286098376944350678155480002493544178052957741 23	1	49.58	97.99
739941636736779718056363618504211299540192687449	-2185483	240	739941636736779718056364919031329645281016361647	1	55.21	101.04
121451520412230388817500321647938483301442594829 3	-2555851	238	121451520412230388817500502373449652880231269644 3	1	52.45	105.39
130230568533273335544123990656933680375751695097 9	-4826635	232	130230568533273335544124061886939420302780803537 9	1	48.98	106.58
121618303251914887984858155112233188200850807494 7	-2503147	243	121618303251914887984857949699974699560636707579 9	1	57.45	109.58
142260621220657378472284163895996858219253678311 1	-5668147	234	142260621220657378472284370680198715565491574430 1	1	56.05	113.14
135798869100804949313321130139292145150431353449	-1481539	243	135798869100804949313321270744238515869157237256 1	1	67.6	113.76
144895897865893293341494196880707301090283835593 1	-4362835	246	144895897865893293341494341184267787519654821634 9	1	60	122.25
101321507065970135228738290737411514642708903789 7	-3666811	254	101321507065970135228738226505951696065893603583 1	1	65.4	129.61
134453060055991522234070839205916265410398926320 7	-3885907	264	134453060055991522340708022549032409067224637389	1	71.13	145.11
92869905606703916716439722049611716734094709794 7	-6009187	264	928699056067039167164397898482022792679250391229	1	84.68	164.5
871308262160246007610132600917628945317611912333	-5973307	268	871308262160246007610134431987315367199106197417	1	90.81	172.88
132236993539573931436232049899396780478502392584 7	-4198963	319	132236993539573931436232262439174552820358165194 9	1	139.79	259.27

p	Δ	h	$\lfloor \log_2 r \rfloor + 1$	k	time of findDiscriminant	time of cryptoCurve
1142427043951443703415867166483898484171097425286359690002762093054331872951	-450059	200	250	1	0.44	42.52
1768439440945156973866092678113238289726465688819081141758094977490355715151389	-1339795	200	250	1	2.37	46.56
952092709588948826716436985079267872548661852745632584049042895041064228831	-1283011	201	250	1	6.37	49.76
1104414614874616252529247101186349559998028991269030175065096952453035041123	-2063227	200	250	1	1.31	50.98
1137890155600337377359159541131075729410996287902368753284026548098900524647	-1664323	202	250	1	5.93	53.06
1260593224538462281985430878844378709945400698294369324563278771936480877607	-2423443	200	250	1	4.24	54.81
1309619371813967714888873670313399468138334173130554285297322938224462135257	-2476243	200	250	1	3.99	55.16
1020004071363532070847120173631151892262409693159946001189722340357792931703	-866899	204	250	1	11.84	55.67
1479049707008778816139166002852026508925675580140132806492859357517599296153	-3663907	200	250	1	2.48	55.94
1431282239148951292456734603786196047755823195841535906635589299053458686839	-2968795	200	250	1	4.23	56.69
1182509278086868680477497265005115580437738225065367157029537435447870821347	-4188907	200	250	1	2.58	56.76
1412028643041078082242292946171585047038680001782833468874606694220042290259	-2787523	200	250	1	3.46	57.48
1511876412488400387829788741145655474924936523214179825702634558648298822634737	-1770907	204	250	1	10.75	61.8
1600113476832985322551133795378036562237232328658648907658449663730433987627 3	-4597723	200	250	1	5.1	62.87
1384513608618493189830218377463011333941162451579985722240887139529574436019	-1145971	205	250	1	18.24	63.32
1225665186328842274561947783591498051607876167039345936225694584246681685813	-372731	206	250	1	19.79	64.64
1141214967892678884922800656979491252055783485249837821245156466573740316799	-4683427	200	250	1	8.79	65.61
1430308567123307424787979786031998306064277055408687231174669133968885589836111	-914059	208	250	1	21.94	67.08
1194361895678878827237935801542203683919381410372115242242565236838464349743	-1175251	208	250	1	24.28	72.86
1567374784340583134077057965789600017524970587221317804117233922226080483053	-3052723	204	250	1	18.94	73.14
1270774382849206394910282465556271925639405305923005793932533580682350714227 1	-2172835	204	250	1	19.03	73.49
1758456061486214366278094089121021370185183208515602894672175238009149466079	-3689683	204	250	1	19.76	75.15
1331390399317439174182503606533674616928828923313329373378902293425633453	-3076123	206	250	1	20.26	75.5
1541742236259398544444950734805881884409166319501658016928200688424492528703	-4746883	206	250	1	20.84	78.06
1133213609593029801752547813399283394172987902531734215166933017611982114931	-4985467	204	250	1	19.59	79.43
1567519003875191717328830634142736116505566235860874647201293124740252056191	-1435531	208	250	1	30.36	80.47
1087596395037802287498728614747953084531212694378831310381288310373865632917	-2308363	208	250	1	25.73	80.7
1171221939893724512445803848630837133562275501490039464106859261596920202253	-3701107	208	250	1	27.97	81.57
9880140131772767891768396413684109542506987468467067114497080348324370888 39	-2155915	208	250	1	24.85	81.71
1052438388863253734659395764444717902383492120435545078805218955392722533037	-2912083	210	250	1	27.77	83.17
1159119749144225140008716745905651197408459251404673543092255319779023597927	-3623107	207	250	1	24.47	83.51
9324905690049031660072473505189124868449456349724364566622246221180096555 81	-2088235	208	250	1	31.1	85.87
9197191537272660004795320500502804743698909316062029459250120377693887062 49	-1614787	209	250	1	37.75	85.97
1567723117475715748032591505020830508832758754312823657785252131577665290 7	-813971	211	250	1	32.97	86.86
1523439878116882719306152189588561613278052927114465849636453137204677126817	-1483627	209	250	1	37.25	87.53
1655430276683429606716434976214896181383644982559897893666352686429191508163	-2486707	210	250	1	37.37	91.55
1165768188975608982387407075968369006224814201936119759084549673033294224589	-4983595	208	250	1	31.91	93.2
1749910932170599201698472666896560892331814060100385028435012101011695184247	-1362931	213	250	1	44.07	93.94
1239611605632491410493097950052071401851961729908392883599812933129821698231	-3850867	208	250	1	34.77	94.34
1635434271707818716419702554340905704962923643311944139031398408013433028397	-3863083	212	250	1	36.05	95.67

p	Δ	h	$\lfloor \log_2 r \rfloor + 1$	k	time of find-Discriminant	time of crypto-Curve
157633342558601298346506931983570154619323429028799999252068679695662595212 1	-3564283	209	250	1	39.98	99.76
92524403856176106063426680006789817788273872306266470342940066659421721690 7	-3286963	210	250	1	47.21	101.76
13458095987469040979591908769706075254488004436410382289807623660129567589 21	-2548795	212	250	1	46.69	102.33
12796042586082428488318020255401838579331596657673288931224275819899710101 23	-2349763	216	250	1	47.74	103
16905450426160119413478498801677823860361303378377205960785422137046851077 69	-2222947	210	250	1	51.39	105.57
13762569661272750879612702632092891663026535606917023459594491566822720715 67	-2974843	216	250	1	51.71	107.53
17652400988546734824296326488991517454774064515455647476721784060732452942 19	-4512307	212	250	1	47.49	107.64
12705257564249624110067023744480966028590193338454273878638595236584754071 381	-3318883	214	250	1	52.89	109.53
13483199557072380482760948770256745368556361896563022328718432131665493057 93	-3615403	216	250	1	50.42	110.2
11762577864722673649012846529692133653131111714330848369732085342541899163 01	-3448003	214	250	1	53.11	111.48
11536864249834207592495007322243243983742372555865661308635899724661078662 91	-2770891	213	250	1	53.78	112.32
12576721949294454903704124520391580297776134813539998111784331468082854360 57	-7680830	218	250	1	63.14	114.42
14276560607708916903600259318395682243132695175719471256173381695131889289 43	-3849307	216	250	1	59.1	118.6
10361918229243958395475751916152441543463290931783038064132147570245008470 83	-1867843	220	250	1	63.91	119.39
11508034859782972480840725899397498301936272663256794973928001859573293420 61	-2606203	214	250	1	63.42	119.55
98250037610888566759492586379100285624853899364491078010980835533971952411 7	-2475403	218	250	1	67.43	119.87
14555427474061731421416655769626193567728202767435417102944227267092793166557	-734147	220	250	1	70.72	121.66
14058213691476532617884638062141095653798125390372500615905573741004804427 11	-3873907	216	250	1	63.67	122.73
11260289036328836041353603517973725586390543011688436976935268594507652046 67	-742763	222	250	1	78.05	130.93
11167790822812515900097054460008693400153896204601462386047084769948242166 79	-3591163	221	250	1	69.3	131.5
17673902568483891067013062962224819031741117137024936417856947504501843000 41	-1530955	220	250	1	78.48	135.98
12527492206176122764436225160666889120140869983520002823124149271055901280271	-1415507	217	250	1	80.46	137.16
93831833991179355359959766845202510982281022441238204198095581654562971331 1	-3898315	222	250	1	80.7	142.69
15839863902910718575845367623019005148369562081240423489122120754868772940 39	-2637787	221	250	1	85.31	143.06
94655773392963562936718747941417724377678494455581069607770451108663506291 1	-4478443	220	250	1	82.84	146.29
16132304387160984278184332285061295561134143493820243320472193492459180788 81	-4141795	216	250	1	84.08	147.91
17676861826110883887315460569614161864059826651730971070016345939294091397801	-1673155	224	250	1	90.76	148.58
98358281492675650001541583723047718842716963536194240965549009999030954855007	-4546747	234	250	1	93.03	163.71
91234908467144609393731486223593746861144631435680629321889856221559229189 31	-2417755	228	250	1	104.3	163.86
11736824555675403016047464066492558514010455255651102808213001024625786572041 1	-3230347	225	250	1	110.48	171.37
98119968691038384862611592012331894831384130940141606787161225391224607942 3	-1031659	236	250	1	118.81	174.71
13242423929866219585935806088148125433027477794990665758851652047165759885 23	-1346539	233	250	1	121.94	180.85
15770773476607515994228270654100917191733560046152692468416519596282321830 37	-4306387	240	250	1	108.92	183.9
12307470072951612480471179877779242893502941726547987767140235903708861058 09	-5098195	232	250	1	125.28	201.38
11921593277715379354552828071181304365963645347483602230144325378635063190 87	-3646123	240	250	1	131.71	204.27
17019782764310145485861551676095860893241227291297371525136261752594760505517	-3109267	238	250	1	120.08	209.81
10989228119417476852451350920800537514177048398406400085401690567252431925 93	-755819	252	250	1	159.23	225.1
11546510354401000202133006578816825392129604264077447441575320599001444966 29	-4630723	235	250	1	154.07	230.78
15339353451338471017195228121859623163215510907457706662661033889294515 23	-2795707	254	250	1	167.49	245.22
11559248510432248316234404310942778365986299261364482213107838508785857872 43	-2961643	254	250	1	175.55	248.96

p	Δ	h	$\lfloor \log_2 r \rfloor + 1$	k	time of `findDiscriminant`	time of `cryptoCurve`
13794675386974273378557619976149305473467356410539283712826487594108913748 01	-5923747	238	250	1	184.62	264.8
15010936323238656230501479548035808704438937752818177899133242520960620828 53	-3361723	260	250	1	176.88	268.06
17329479420434709387310679924510697299714677844745126395364994031796502598 11	-4882987	257	250	1	178.79	269.59
14478492491636621828435427750013229648146258455118552581195468208655017730 39	-3096955	260	250	1	188.98	282.82
15337418600183353746155733950856713968250331103523015945996865642985496777 63	-1854691	276	250	1	221.48	320.68
15230688449947169560470597883765544804071392735762143441556441825260694230 57	-4345003	268	250	1	218.64	323.43
11863143004498886895284401934590146060316613925296748458673231417778807650 11	-4224787	260	250	1	227.37	329.73
10726595214521204160715441958958798688352232376566556568706037942456071808 1	-4547323	260	250	1	268.68	365.59
10280557170075143062050815890007901895794198149806122871403621722725697827 53	-5315587	286	250	1	234.64	371.05
12705171809009031203647689350333960182764036453240791284665501903132964364 53	-5860603	280	250	1	255.05	379.33
10153204932036576762931216669642930177215438999060471968124467124335696396 23	-1343963	287	250	1	264.91	380.12
13525073923664212745977895647364136821375522761492854021388594781908080138 31	-2618203	266	250	1	313.85	407.64
14617533888856398065119383888152899239173439444549397256375685223122497807 11	-3557731	276	250	1	322.38	430.96
17285297092053643106788986540519154039107377466249652291873967605593976360257	-4056019	292	250	1	325.12	445.4
14442381488853292839537564058793938901416050121851706787680392296757515716121	-5544427	276	250	1	327.09	454.59
11789135077531305511161670723153282583490319677600983405725670611266173177 87	-5504203	322	250	1	313.1	471.61
17311286493153030401131062019943434549006941819120004742246108180830065303 41	-3314203	315	250	1	431.75	567.21
14715187158497972563774682161682080394905508079999252124338404026269496972 93	-4453147	363	250	1	430.37	622.57
16121122277784385714253857125789072511828337105524970849689523009522883852 53	-3211291	359	250	1	491.03	684.77
17446745723498784638989315339210202157080026409659696727387179537547532739 57	-4797227	519	250	1	804.94	1264.46

A.3 Sample Large Fourier Coefficients of j

We list the Fourier coefficients c_n of the modular function j for $49996 \leq n \leq 50000$.

$$
\begin{aligned}
c_{49996} = {} & 39767056188679581370703503676951007582680592251895221989079476182289462188843189294 6 \backslash \\
& 65371929854387559146232995559900908900177604034747063632964335793104001085238564089 \backslash \\
& 51634993938362066932768261194613289443998212297421987157005510900585702876107833667 4 \backslash \\
& 21842249679812472663077800792191446214548451121911957870951752407062131228722410049 7 \backslash \\
& 84242566679643945887782004584923562171533233901108755884458402801432890189903676319 5 \backslash \\
& 10641057876657961577907623407501999835474060125632353444010853361572466794229434750 7 \backslash \\
& 53799241036054377920836722233404995740134710826612688643987122864137014389698817903 4 \backslash \\
& 22398049431153195610435001233690902257810004279922321649097940378329598412829468867 4 \backslash \\
& 93590019073972592773061246249862677461442426441982049258090144443431320434024551554 8 \backslash \\
& 64124226340354714715666072159838360235653414159590612221585977994942963779772186460 0 \backslash \\
& 06242385643999355182457439034057934961326215483969268543965730894359341145643292170 8 \backslash \\
& 62304501243245576736314084493742223125833119442751676592102208647979755301438126762 2 \backslash \\
& 85553587485162454351251323285674565665160395628780760331339926338350357930191449902 5 \backslash \\
& 05663933614698410507654022111126069456236692111464147217230884774332293595370865464 3 \backslash \\
& 84235626652684816605094503561201554869800 \\
c_{49997} = {} & 40899766706638725107463274111257339086590100974288572043759110903916644067010386754 4 \backslash \\
& 84513118690920941412051432389065060448289221416024799962650911566460298947757964862 3 \backslash \\
& 30488615174361109825940264177195505310207962886154524829014599188480546917346570305 7 \backslash \\
& 09092310492256884893366342908785781100813789061486232891715110747278804117106066864 3 \backslash \\
& 61675921780052539218273122496980947080655625210294490667166255997075374912533601034 1 \backslash \\
& 51434739950111715420906830728471187022301305232987503225809056045070542285772926997 4 \backslash \\
& 02512827788200967402560180350290651869191825793283887571191028968823651143475589305 9 \backslash \\
& 48841383208780030179627541641667004823134058274144155331213504575675903271547373550 7 \backslash \\
& 22657690656063062180284338830836825458897363232458876441379192946628693692242830984 9 \backslash \\
& 61805817507132329071399829724736529385055585591137405970747214055234021821422636474 4 \backslash \\
& 73964976637360907459064859081714686033528108386189867355662117604348658730088698295 1 \backslash \\
& 27483767341821528710263784485273496323999792306370496849998182658190013073982146864 1 \backslash \\
& 03926068546691319652290132574052707884796949193409363399641422693828446862038700646 3 \backslash \\
& 08307486888426246562900614262899744939294106245064382865271499901367392778227277391 7 \backslash \\
& 66903872937241525819614004034763230806016 \\
c_{49998} = {} & 42064729134698149237252939545524720686796035390191052688195583206180196689930220165 0 \backslash \\
& 94282087691682303021790473046993602043093874294742384070070516484643174383066940728 7 \backslash \\
& 73788802204545809070002862038006727842993798823921747158606401687976475055598455051 2 \backslash \\
& 43489946435456070358890935493248633113502637746494123719579309296804392656491585275 8 \backslash \\
& 59650956746302003012792179161198705372956719886443826305500165206510412845000804092 5 \backslash \\
& 73261059269182433977405854099290535563719131189198203148831757255159996720649346528 7 \backslash \\
& 78719348408349308898473159612074785603060992759594127874603407552949930897798922614 6 \backslash \\
& 75390898158270473259439995036293735823033254558037267846062541857933224584250705218 8 \backslash \\
& 27868614805604488590060597914944320984379458845174586682106400460661737149903499985 3 \backslash \\
& 40456068077938803908887126231619539290034105318133923875836838070248165912565462440 4 \backslash \\
& 85286633426667714858123474892035466764952825875200507286754162668855746946717453983 7 \backslash \\
& 69927696733189438405838517651420787871041086947394438889366162934661418223415641920 7 \backslash \\
& 22440041389010404333202053623929890634383722697407695972276885275288383626820053785 1 \backslash \\
& 04129552518251497421363857112114285510698547951533360021245924927992460685287068555 5 \backslash \\
& 55114120926506462163083247925569595425960
\end{aligned}
$$

$$c_{49999} = 432628614523252445646788383366867521393587537454952665168740174359700485742158475245 \backslash$$
$$618752965353664962736254160239073445503865513981676164910463402258809923681356079148 \backslash$$
$$304773284277479813927481311789004006124672351636649833785051518209626861912487268850 \backslash$$
$$019743261405325910586383456127421450728118638249148331865296690159574060709985731773 \backslash$$
$$248694631015242055982119507234173987726555325870706083738894827945734813458151806119 \backslash$$
$$101028195732025203843621813301704152055122925958550895270982507332153438000539899628 \backslash$$
$$841542334433810337688691084512432187925988393028755559576083148226024662995096548302 \backslash$$
$$871247659401904561465613838715721818569721004153136498134979775500072580062188637174 \backslash$$
$$103114229141902957892295430478504116952658539240893362338242932132526593051484870820 \backslash$$
$$559617943666223247717691667433012449707154926288049943709366001561614790215002183384 \backslash$$
$$422735922309164877767278212766304031761260036845240320916153676025099330183101317799 \backslash$$
$$558856439105152211509556265816032806263598710562398629279346993272020757738737565256 \backslash$$
$$981333768097937156577037824302860202658355597149325037553909913654776629685647197130 \backslash$$
$$455865395756591367943029791534069114887468883028117851789785197029219353570824843463 \backslash$$
$$021002099641166274150166260238785686732 80$$

$$c_{50000} = 444951077576849371688885934203367408199138569813259772117759330043215997817429361996 \backslash$$
$$667338988736544528333902679169320974586757302005306538597510275448708437846696770448 \backslash$$
$$807070480357711167205736679700553877861898752461486939605965262035749250581316542629 \backslash$$
$$012038668551314031492733241151694016551154633521355413432221429759883495237548933153 \backslash$$
$$657683605228948628744877197444055772568576981762838140802435883732893181190882498657 \backslash$$
$$908655764710995267409458765441602298504120928712983920154939736943013347558268165782 \backslash$$
$$839492952939836554727016106091554413950755124586737673797365245147250934704752637481 \backslash$$
$$755550688413756546044889437045996682459650147187192112964931859651124081936348183178 \backslash$$
$$946694806776098265930030101267989966233197025750730039284640690539658418760546044012 \backslash$$
$$837166794127581318705063765426065191952523600405470357695667452195491646389913275602 \backslash$$
$$909969538143906539403527448982438886502294161630978759278252064434632356633444177527 \backslash$$
$$133140995375163682146161277450639761150751801250464769790047333811760697090959668621 \backslash$$
$$658047807751768082824547620445171874597101285833185648886831838171461999868608567665 \backslash$$
$$026781211766568690823662638456282942171550439866335050669681368578723001350020027882 \backslash$$
$$628388511743146073057984738478827928545 02$$

A.4 Running Times of `cryptoCurve`

We provide practical running times of our generating algorithm `cryptoCurve`(r_0, k_0, h_0) for various input r_0, k_0, and h_0. We use the Fixed Discriminant Approach.

$\lfloor \log_2 r \rfloor + 1$	time of `computeClassInvariants` and `computeClassPolynomial`	time of `find_root`	time of `cryptoCurve`
160	0.63	10.2	11.09
170	0.63	11.99	12.86
180	0.63	13.61	14.51
190	0.62	14.33	15.25
200	0.62	16.63	18.17
210	0.63	17.55	20.39
220	0.63	19.43	20.38
230	0.62	22.21	23.76
240	0.62	23.05	24.95
250	0.63	25.12	27.86
260	0.62	28.26	29.64
270	0.62	28.44	32.29
280	0.62	32.98	37.27
290	0.62	36.38	39.43
300	0.63	36.92	39.01
310	0.63	40.52	42.52
320	0.62	44.99	47.12
330	0.62	46.41	49.97
340	0.62	49.21	52.03
350	0.62	51.31	55.35
360	0.62	56.13	61.75
370	0.62	59.87	64.95
380	0.62	61.48	67.82
390	0.62	65.13	72.67
400	0.62	71.42	78.75
410	0.62	70.11	80.61
420	0.62	78.81	89.31
430	0.62	85.37	97.85
440	0.62	87.05	101.08
450	0.62	94.03	108.43
460	0.62	100.25	114.13
470	0.62	92.05	118.57
480	0.62	111.58	121.64
490	0.62	114.42	127.82
500	0.62	113.86	135.81

Table A.1: Running time of `cryptoCurve`$(2^b, 4, 200)$ on the Pentium III for $159 \leq b \leq 499$. All timings are given in seconds. The discriminant is $\Delta = -21311$; it is maximal and fundamental with $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$ and $h(\Delta) = 200$. The class polynomial W is computed during the algorithm.

$\lfloor \log_2 r \rfloor + 1$	time of `computeClassInvariants` and `computeClassPolynomial`	time of `find_root`	time of `cryptoCurve`
160	3.43	26.77	30.55
170	3.5	31.21	35.52
180	3.45	35.49	39.33
190	3.45	37.89	41.89
200	3.43	43.23	47.96
210	3.41	45.58	49.6
220	3.42	51.77	55.97
230	3.43	56.67	60.61
240	3.44	60.2	64.23
250	3.45	66.62	72.03
260	3.44	77.94	82.5
270	3.43	79.92	88.9
280	3.44	86.35	91.89
290	3.43	96	104.13
300	3.46	98.48	102.52
310	3.44	109.64	115.09
320	3.41	116.27	123.85
330	3.43	125.65	130.48
340	3.43	129.86	135.97
350	3.44	141.29	149.32
360	3.45	143.58	157.56
370	3.43	155.35	161.26
380	3.42	162.89	167.59
390	3.41	176.74	195.71
400	3.41	188.87	198.57
410	3.42	184.3	193.68
420	3.43	211.93	217.47
430	3.41	219.49	229.36
440	3.45	226.91	237.22
450	3.42	252.16	258.32
460	3.46	258.59	264.97
470	3.45	259.28	266.97
480	3.41	290.51	311.13
490	3.41	296.55	317.22
500	3.45	292.59	333.04

Table A.2: Running time of `cryptoCurve`$(2^b, 4, 500)$ on the Pentium III for $159 \leq b \leq 499$. All timings are given in seconds. The discriminant is $\Delta = -96599$; it is maximal and fundamental with $\Delta \equiv 1 \bmod 8$, $3 \nmid \Delta$ and $h(\Delta) = 500$. The class polynomial W is computed during the algorithm.

$\lfloor \log_2 r \rfloor + 1$	time of `computeClassInvariants` and `computeClassPolynomial`	time of `find_root`	time of `cryptoCurve`
160	18.49	9.88	28.84
170	18.48	11.87	31.11
180	18.44	12.55	31.53
190	18.41	13.72	32.77
200	18.41	16.2	37.28
210	18.4	16.73	36.92
220	18.4	19.4	39.75
230	18.4	21.46	40.98
240	18.41	22.94	42.09
250	18.41	24.99	47.07
260	18.41	28.04	47.13
270	18.41	29.13	53.05
280	18.4	31.97	52.12
290	18.41	33.93	58.23
300	18.41	36	56.53
310	18.42	39.01	63.62
320	18.4	41.32	69.85
330	18.39	45.77	66.8
340	18.4	48.97	69.14
350	18.41	49.72	70.28
360	18.4	54.78	78.85
370	18.47	59.16	82.72
380	18.41	58.81	87.24
390	18.43	64.59	92.21
400	18.41	69.95	97.87
410	18.42	72.14	103.15
420	18.4	81.19	107.39
430	18.42	84.14	111.89
440	18.42	84.94	115.08
450	18.41	87.71	119.21
460	18.41	92.34	125.74
470	18.41	95.6	130.02
480	18.41	99.66	134.12
490	18.41	105.71	142.59
500	18.41	112.13	151.56

Table A.3: Running time of `cryptoCurve`$(2^b, 1, 200)$ on the Pentium III for $159 \leq b \leq 499$. All timings are given in seconds. The discriminant is $\Delta = -125579$; it is maximal and fundamental with $\Delta \equiv 5 \bmod 8$, $3 \nmid \Delta$ and $h(\Delta) = 200$. The class polynomial G is computed during the algorithm.

A.5 Class Polynomials W of Large Degree

We first give the largest coefficient (with respect to the absolute value) of the Weber polynomial W for $\Delta = -55222439$ of class number $h(\Delta) = 15000$.

$$
\begin{aligned}
w_{3862} = {} & -534016897550477416526384430031521462583184171000776121893609507858738823415099336 24 \backslash \\
& 904564281484988174924156527826456816983405285272301282745085562902377948041854897259 7 \backslash \\
& 924658464982172291597699980851356428421124910995253452453976071430051896203737603010 7 \backslash \\
& 366108970487723032210867500323763983632306241493558191448295856249876680113096315968 0 \backslash \\
& 322914680888663801429321320063861012832326630280192126270764378261576617886922672875 6 \backslash \\
& 001328969583748772070957343096476530045180680205766841858146404758938715119217860468 1 \backslash \\
& 561682161229832017859536178490100093428369271737944520890566822407347863967004803713 4 \backslash \\
& 842619197589744055923303140264870956647040609221709621416510336229310331824862358832 9 \backslash \\
& 040097659409538211449771943596705835691793179409643671318875974562420861245403390298 0 \backslash \\
& 498155624959309249721191186133399036916614568130431818495006973410884018118527809924 6 \backslash \\
& 394329219973178444748079515895574742569509520638988254246291360058546984344058947133 4 \backslash \\
& 981667296094345988604106189036156871789048181348447301128661699156311986328218735194 5 \backslash \\
& 574653451639729079862397908696169619020143093119938896763125143391895742334735039558 1 \backslash \\
& 388850281788288201127054827136397305091008444080610388397770774022495817750640771791 3 \backslash \\
& 522866111712585914385309059575640167727364690106589722701664083274665955377456839249 7 \backslash \\
& 604685203048635359677345200774795681427949604225399737115292225351596632778878019748 4 \backslash \\
& 524608130057652049595815215151494840676862728407758636524363683096123249013523129299 9 \backslash \\
& 410539265613476901302482487780911857882281394214635819216678623362722193853339739663 0 \backslash \\
& 351934769792857913887843964161409577316414255445339631824791319500094340464294050704 0 \backslash \\
& 870512129839761109805833836582238057100351288067996466896626023274050011691304135454 0 \backslash \\
& 177786576228208618198468858547225768377470591701896923530431203228596419209990122591 5 \backslash \\
& 034321624480862934974487799187942037873896192176043052072166056229585420040646249519 6 \backslash \\
& 460371177423873122426643627175213159666308127623242174115048879214619680357520741362 6 \backslash \\
& 753534280165474217343039550713403414759487970493767602454356278110399191951606011485 \backslash \\
& 574173490337717516833701150919361244529057735232774415816688533788022787906230815082 7 \backslash \\
& 524859605548836886015734832437471582316250836419579504602464502204160034271226777550 2 \backslash \\
& 547320024497791010071797166447862310268297203799953895406602268656120574387250790064 8 \backslash \\
& 289209376664945710398834900918221615933396967360200908175197545372188252173055330593 1 \backslash \\
& 182372544179532048173633730229433339551789093040955119256152135329569914864212943131 9 \backslash \\
& 459659839965164188573862351684236964453204768875694533505781615076331796172413905501 7 \backslash \\
& 752636382983205564459116483400485802890165330928556982999475133251187749370123508990 \backslash \\
& 702937191852436071147787621075773621079843253985185748841729186262243877812450873601 1 \backslash \\
& 620927161132052948766999681257801534900036207266714238483341160001270429865007467473 3 \backslash \\
& 428062606147283314401995036329673326531887011141929569032459732616002256536882870718 6 \backslash \\
& 432676126935222134039903984925371783751477887866013017813751598958021682127225957042 4 \backslash \\
& 270647355649952411727716519014259009762249433219201319498652714409554522380790217324 8 \backslash \\
& 372864792878284173377398359537851865259984383959700973493981850839435594329362453058 2 \backslash \\
& 485369616774376554469163184625679874105284761567121578786580840739525506489845385263 7 \backslash \\
& 246247317672342117596317842891748651925868348993675237703385031650822593344556634378 8 \backslash \\
& 236448676426001640188140230565960191328811975701932288194842633158631156506277905320 9 \backslash \\
& 684588223427060392198798791451885085074755512746130128843157829381314784461632477793 0 \backslash \\
& 837096031215366062953751390802918676382808119814186405257462512935773088208762461959 4 \backslash \\
& 799870894022
\end{aligned}
$$

We next present CPU-timings of our algorithm `cryptoCurve`$(2^{159}, 4, h_0)$ if h_0 is at least 3000.

Δ	$h = h_0$	time of `findPrime`	time of `find_root`	time of `cryptoCurve`
-2668511	3000	0.2978	265.7	266.78
-3670631	3500	0.295	289.91	291.25
-4728671	4000	0.2718	310.92	312.45
-5899871	4500	0.3486	513.7	515.79
-6961631	5000	0.2448	533.71	535.99
-8666831	5500	0.282	558.96	561.5
-10112231	6000	0.2788	584.54	587.34
-11658191	6500	0.311	617.49	621.19
-13090271	7000	0.2502	636.81	640.44
-14796911	7500	0.353	651.41	655.34
-18039551	8000	0.3014	678.66	683.45
-18365591	8500	0.3096	1097.48	1102.55
-19829039	9000	0.2496	1113.04	1118.38
-21649151	9500	0.2046	1151.41	1157.24
-23512271	10000	0.2418	1174.46	1181.36
-26589239	10500	0.2054	1183.68	1190.54
-31624031	11000	0.2084	1220.19	1228.22
-36419759	11500	0.306	1258.29	1268.22
-37398479	12000	0.3948	1295.38	1304.82
-43263359	12500	0.4314	1300.28	1311.05
-42631391	13000	0.2472	1332.79	1343.56
-43531199	13500	0.296	1332.51	1343.44
-46429631	14000	0.308	1392.67	1403.91
-50279759	14500	0.3652	1421.89	1434.75
-55222439	15000	0.4484	1400.18	1413.42

Table A.4: Timings of `cryptoCurve`$(2^{159}, 4, h_0)$ for $3000 \leq h_0 \leq 15000$ on the Pentium III using the database classPolynomials. All timings are given in seconds.

Appendix B

Sample Elliptic Curves

We list various elliptic curves suitable for use in cryptography. All curves in the subsequent sections respect the requirements of Section 2.3.5. Hence, either curve is in conformance with the German Digital Signature Act. In addition, all curves are of the form $(-3, b)$. Elliptic curves with $a = -3$ are attractive for software implementation of the curve arithmetic.

First, in Section B.1 we present elliptic curves of cofactor 4. The endomorphism ring of each curve has a discriminant congruent 1 modulo 8. All curves are defined over a field of minimal bitlength, that is over a 162-bit field. We distinguish the case of a small and a large class number of the endomorphism ring of the curve, respectively. More precisely, we first list in Section B.1.1 elliptic curves whose endomorphism ring is of class number at most 1000. Next, in Section B.1.2 we present curves corresponding to class numbers up to 15000.

Second, in Section B.2 we present curves of cofactor 4 which are defined over fields of different bitlength, that is fields of bitlength up to 502. In Section B.2.1 we list curves whose endomorphism ring has discriminant -21311. Next, in Section B.2.2 we present curves having -96599 as discriminant of their endomorphism ring.

Third, Section B.3 lists curves of prime order. Again, we consider curves over a field of minimal bitlength, that is fields of bitlength 160. In addition, we present curves over fields of bitlength up to 500.

Fourth, in Section B.4 we give some examples of twisted pairs for various discriminants. In addition, sample twisted pairs over fields of a bitlength up to 500 are given. All elliptic curves and their twists are of the form $(-3, b)$ and $(-3, b')$, respectively.

Finally, Section B.5 presents sample elliptic curves of prime order defined over Optimal Extension Fields suitable for use in cryptography. The curves are output of `findOEF`$(32, 5, h_0)$ for $200 \leq h_0 \leq 500$, $50 \mid h_0$.

B.1 Elliptic Curves over 162-bit Fields with $k = 4$

B.1.1 Elliptic Curves of Small Class Number

We present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is of bitlength 162. All curves have a cofactor k equal to 4. In addition, either group order is divisible by a prime r of bitlength 160. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is congruent 1 modulo 8. The class numbers are in $[200; 1000]$. Either discriminant is maximal and fundamental of a given class number.

Δ	=	-21311
$h(\Delta)$	=	200
p	=	485115340538830080774701226497093234829238249432 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	1212788351347075201936753488952938384214999551721
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	485115340538830080774701226497093234829238249432 6
b	=	347048043946807090919733503379345928938915415846 5
G	=	(238615183755145351025139150128834676474930665606 6, 444076929318501068033331845592892397993596828292 5)

Δ	=	-30551
$h(\Delta)$	=	250
p	=	486547956384460881033014550642711415864989066912 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	1216369890961152202582537089915532627525193151061
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	486547956384460881033014550642711415864989066911 8
b	=	231391162601242956557803159216955366091125510701 2
G	=	(211133114307925698303596366778909960039147961459 3, 716756175456003302230004619141754783175930540742)

Δ	=	-34271
$h(\Delta)$	=	300
p	=	403546989521402932869665667915957447845264522634 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	1008867473803507332174163682727528543154984279171
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	403546989521402932869665667915957447845264522633 8
b	=	375268319981340986519650576838220625023374989833 8
G	=	(471947776607623737496386700021735083783400650076, 235435433253096179249627788415422036635058449740)

Δ	=	-47759
$h(\Delta)$	=	350
p	=	461568204972514951680415634682627614021960463774 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	1153920512431287379201038744199812555021860892871
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	461568204972514951680415634682627614021960463773 8
b	=	234970424385695966193255651561186136547318835746 2
G	=	(176629963568403251242884946252671047006781158071 9,
		171703491022532908643141938142401393378631199223)

Δ	=	-67031
$h(\Delta)$	=	400
p	=	354023413742495631182261569965578207380321175232 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	885058534356239077955654317113163030774648373761
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	354023413742495631182261569965578207380321175231 8
b	=	750261320355185714401223457564399210484430461766
G	=	(413814562411521608412745268111846710084954405896,
		107329200946240287588224203915037551040904256816 9)

Δ	=	-75599
$h(\Delta)$	=	450
p	=	465497151589650352042087302679588944926092020492 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	116374287897412588010521784103952532973972644646 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	465497151589650352042087302679588944926092020491 8
b	=	241658859363724241329704130650469062568421767098
G	=	(55799888553902317492988064736525033599653068844 0,
		31070778555520273691339995560554839298238203728 1)

Δ	=	-96599
$h(\Delta)$	=	500
p	=	29277705022189437915158835195832187352996584887 69
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	731942625554735947878970388457517964619739358449
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	29277705022189437915158835195832187352996584887 66
b	=	27891497239875238325565515455155018507926146390 75
G	=	(22600130735015459877832185679641909879774962901 22,
		63806018095672659709551091859616871728799975292)

Δ	=	-148511
$h(\Delta)$	=	600
p	=	53827888251935664269546166942130957390101651643 81
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	13456972062983916067386542941157015424439498197 91
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	53827888251935664269546166942130957390101651643 78
b	=	41924563671642948214313482372266358891847444428 83
G	=	(32638395958415407315072867487854817578864638219 75,
		18823254103797677662959770262481931971800963634 54)

Δ	=	-185471
$h(\Delta)$	=	700
p	=	50140720272672864893262376409934532913182095679 09
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	12535180068168216223315599101829429973130957379 79
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	50140720272672864893262376409934532913182095679 06
b	=	15015909229661156738185460350335818700061080183 45
G	=	(11143011216810529058262637679933885240838896826 8,
		37316410111920550572449382835922443629901314906 12)

Δ	=	-233999
$h(\Delta)$	=	800
p	=	577451580959996769517785605603429333558904652608 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	144362895239999192379446386886234855025171663474 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	577451580959996769517785605603429333558904652607 8
b	=	521925673338314051136544198647137848973966385912 6
G	=	(411744531196010226472061593923606399936890202786 4,
		118384770291599286800186536639515944950559122654 1)

Δ	=	-299519
$h(\Delta)$	=	900
p	=	430335445751677189064443120705863441083499561478 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	107583861437919297266110824790392940650235996053 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	430335445751677189064443120705863441083499561478 6
b	=	460872319528424852164549979963852943157512903945
G	=	(753616462846189765621178704448388880010221188775,
		342969474079169470116202569720920641581200781563 0)

Δ	=	-412079
$h(\Delta)$	=	1000
p	=	407326443860670664785649396923177411830925477802 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	101831610965167666196412312268764759413586369567 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	407326443860670664785649396923177411830925477802 6
b	=	401482724635668048419652143293375154058612939451 7
G	=	(226402245472055130417094576154222286768603763743 7,
		324729336615386354884071855875535778242733288266 7)

B.1.2 Elliptic Curves of Large Class Number

In this section we present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is of bitlength 162. All curves have a cofactor k equal to 4. In addition, either group order is divisible by a prime r of bitlength 160. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is congruent 1 modulo 8. For each integer h in $[3000; 15000]$, h divisible by 500, we choose a discriminant to be maximal and fundamental of class number h. For either discriminant we make use of the precomputed class polynomial W.

Δ	=	-2668511
$h(\Delta)$	=	3000
p	=	534483831320786971389536389568771733285103072236 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	133620957830196742847384050525532405332807471668 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	534483831320786971389536389568771733285103072235 8
b	=	448139286740243781178817743498730462397491810960 3
G	=	(473381464332156551513420563846882445433500101689 9,
		390986711839392903377840015736995596418391504117 2)

Δ	=	-3670631
$h(\Delta)$	=	3500
p	=	410321415914906630938021691420746098253233102642 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	102580353978726657734505335315826486377141292441 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	410321415914906630938021691420746098253233102641 8
b	=	325953736680743096293846715518159835361310260326 7
G	=	(326697916114295739607867084319955781658769973083 5,
		996308498959166712027104773682000602102811927068)

Δ	=	-4728671
$h(\Delta)$	=	4000
p	=	48390463352415625412612079415506741181774767193 6 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	12097615838103906353153018352089074509417159717 4 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	48390463352415625412612079415506741181774767193 6 6
b	=	24375756138993934055708945929947491440969285220 1 5
G	=	(196841244787247428075463879828054487914007267986 5,
		405580073359720217465826024148449155320378524936 0)

Δ	=	-5899871
$h(\Delta)$	=	4500
p	=	352411790730239313312206078760877321048649637924 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	881029476825598283280515767850347827211219566421
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	352411790730239313312206078760877321048649637923 8
b	=	419866067301764996083119938106562763196618498123
G	=	(232216123903254978370260588610269757498051246291 2,
		225054359999547347076444533992505416026436020509)

Δ	=	-6961631
$h(\Delta)$	=	5000
p	=	346620683659849698096850113866057694679281634576 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	866551709149624245242125683732091208715645000961
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	346620683659849698096850113866057694679281634576 6
b	=	800388995371433774329007961145830277951571279290
G	=	(152809600451457807160552247490537300813220593139,
		339640245482122808978076521226082519061624680974 6)

Δ	=	-8666831
$h(\Delta)$	=	5500
p	=	563117512297128866955229926582278602973671635346 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	140779378074282216738807556136408166625179074991 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	563117512297128866955229926582278602973671635346 6
b	=	250685788947017045156594175632388677255203778576 8
G	=	(493589454859661374109370705948541559624342234980 1,
		514813351634875556430806333648886286125151244070 9)

Δ	=	-10112231
$h(\Delta)$	=	6000
p	=	537873268673272253910017069432836526282100109636 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	134468317168318063477504370857861550534925209224 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	537873268673272253910017069432836526282100109636 6
b	=	488187566877727576889285327674765091897404062853
G	=	(380970769832690628025835151990458611303144983733 2,
		175932560546159688727209688029159734907941827115 2)

Δ	=	-11658191
$h(\Delta)$	=	6500
p	=	344781327260518625233081985109594641839451712618 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	861953318151296563082705272563225798372042385291
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	344781327260518625233081985109594641839451712617 8
b	=	513051049819235883336753536284499396840170789001
G	=	(304465762377942801823151680721220727 4706713301038, 301062218161736901034776007356460536570133091304 7)

Δ	=	-13090271
$h(\Delta)$	=	7000
p	=	480714374636585514014266905326185254383824223490 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	120178593659146378503566790508934012089302909585 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	480714374636585514014266905326185254383824223489 8
b	=	723615305072037058244749862347605552956305273478
G	=	(290111241929941762278827456308168895266171167237 6, 348548202451611419982070772521881350750684071768 2)

Δ	=	-14796911
$h(\Delta)$	=	7500
p	=	506912391841023968368716595300169894893404782100 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	126728097960255992092179064702527214992060680490 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	506912391841023968368716595300169894893404782099 8
b	=	329918555327388196881202278133096111699063772708 5
G	=	(194375463950251269155601386522472955131828549344 6, 420589574066859963842440721504230061265945065746 7)

Δ	=	-18039551
$h(\Delta)$	=	8000
p	=	431492775304918633867704455257137614270815726102 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	107873193826229658466926211759404652883352004681 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	431492775304918633867704455257137614270815726102 6
b	=	247563135698715118888695476629479580952378981648 4
G	=	(726297145421329852982628054946389100931352364611, 229021414260469756023034095244166735850047858067 6)

Δ	=	-18365591
$h(\Delta)$	=	8500
p	=	353378474647482393559063132999203857651618307508 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	883446186618705983897657273598001741029785788681
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	353378474647482393559063132999203857651618307508 6
b	=	273765494134665376772305768771219529050828349469 1
G	=	(130518295745393246148975638090608030414081762425 1,
		197006510423233453449272192804485348155041460079 6)

Δ	=	-19829039
$h(\Delta)$	=	9000
p	=	38603289531475202159740312104427471470161393972 89
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	96508223828688005399350723548227800614859301778 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	38603289531475202159740312104427471470161393972 86
b	=	20048270716357578248402452560786871650861946460 51
G	=	(48351742813118757688592127880899445490043947807 4,
		13585256406498694141742317792600827536894657301 15)

Δ	=	-21649151
$h(\Delta)$	=	9500
p	=	39121709621003883952680968572988071181585231912 89
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	97804274052509709881702515526431155336525441298 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	39121709621003883952680968572988071181585231912 86
b	=	18107733463230663719243262996339260231318950426 03
G	=	(29494234163782283553424120632145300776785446530 57,
		37490019750733556997652024920070352532902066372 34)

Δ	=	-23512271
$h(\Delta)$	=	10000
p	=	49961784180931664247536996773007927296989005323 09
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	12490446045232916061884248682948045460168551709 79
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	49961784180931664247536996773007927296989005323 06
b	=	22611835550579853094952644385737099419625617666 6
G	=	(25090184929379076269950995241088311636653004911 3,
		28665167380866507201530868915574618585740420307 32)

Δ	=	-26589239
$h(\Delta)$	=	10500
p	=	441946658830468035911977654597744839809514370534 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	110486664707617008977994369589313723653225520305 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	441946658830468035911977654597744839809514370534 6
b	=	524347753052919280256584817362587250803884184956
G	=	(246117842889084733081085431439156662827198112808 9,
		406089131972343324220122463926544087905993749180 9)

Δ	=	-31624031
$h(\Delta)$	=	11000
p	=	327527113809386702546245444336078120679886419032 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	818817784523466756365613597711273671182254453721
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	327527113809386702546245444336078120679886419032 6
b	=	209339784291531267576311862963132603611166743143 2
G	=	(815056124095789349480409860532689412876843431627,
		174497157543535697923887785889529380014386020003 7)

Δ	=	-36419759
$h(\Delta)$	=	11500
p	=	547511439202770888536018930933655899942840484970 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	136877859800692722134004641830068514669579030747 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	547511439202770888536018930933655899942840484970 6
b	=	413117092329776184177121413717546731051807206796 1
G	=	(402173797029723966554727684565030505768435672977 1,
		542304911066865389401227095861291971449822935011 8)

Δ	=	-37398479
$h(\Delta)$	=	12000
p	=	389604715925959499151355185187427756835319748302 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	974011789814898747878387330270415716160820807171
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	389604715925959499151355185187427756835319748302 6
b	=	907554658311091317088896231180705989782519430657
G	=	(264536025968750795399561473036198448630526252327 3,
		261750391999579076312032142034394913798910553060 3)

Δ	=	-43263359
$h(\Delta)$	=	12500
p	=	433719197312003773267498122498501042874914760888 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	108429799328000943316874624632357210923209077618 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	433719197312003773267498122498501042874914760888 6
b	=	343908137946029946992865564257795979709025598539 4
G	=	(305481508740312776629817791025995804333988234462 1,
		326863649354842881572812080630838737688592748109 5)

Δ	=	-42631391
$h(\Delta)$	=	13000
p	=	337817291773375653608193127202722417955363322448 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	844543229433439134020481931331269514550771940081
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	337817291773375653608193127202722417955363322448 6
b	=	173287824375058211923901259556675379928847024238 0
G	=	(226248015608718911428893750702450522908342754399 9,
		410067776510981111811881642325125422430682737586)

Δ	=	-43531199
$h(\Delta)$	=	13500
p	=	454766556997744090797073625509460383720786084390 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	113691639249436022699268323800925827239694984065 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	454766556997744090797073625509460383720786084390 6
b	=	159790776075750375914901376761418032692852962294 4
G	=	(452119432210912531244691252625970067232323219637 5,
		359587057047881882875705244160321852442316788237 5)

Δ	=	-46429631
$h(\Delta)$	=	14000
p	=	319831096962105249044396435117639967755281767240 1
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	799577742405263122610991915047533132007957716201
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	319831096962105249044396435117639967755281767239 8
b	=	166101808978025806539969195254399833025781687461 3
G	=	(180133633857951254828318844313724099700372416164 2,
		372848410589282366441416635228752219733160122949)

Δ	=	-50279759
$h(\Delta)$	=	14500
p	=	4094494628923587104286327336368588028591214438801
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	10236236572308967760715811854211508824575167403 01
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	4094494628923587104286327336368588028591214438798
b	=	1202312463471390332831392955644328902319584139223
G	=	(291657914037077568833209155766703567716691964730 9,
		333984901241811835494358465385539526676820011952 1)

Δ	=	-55222439
$h(\Delta)$	=	15000
p	=	467278976722495378301624295428864048779794520816 9
$\lfloor \log_2 p \rfloor + 1$	=	162
r	=	116819744180623844575406031934812604727698450896 1
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	4
a	=	467278976722495378301624295428864048779794520816 6
b	=	241734408954495607893412116784649174438938492767
G	=	(259514056380977384362695117007873984120364539771 9,
		428572489617577650543872933459373693422988209881 8)

B.2 Elliptic Curves over Fields of Different Bitlength with $k = 4$

B.2.1 Elliptic Curves with $\Delta = -21311$

We present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is a prime of bitlength b in $[182; 502]$, $b \equiv 22 \bmod 40$. All curves have a cofactor k equal to 4. In addition, either group order is divisible by a prime r of bitlength $b - 2$. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is $\Delta = -21311$. Its class number is 200.

p	=	42307433392936837192597924432645724879802813461950306 09
$\lfloor \log_2 p \rfloor + 1$	=	182
r	=	10576858348234209298149481099496987032815170335264851 01
$\lfloor \log_2 r \rfloor + 1$	=	180
k	=	4
a	=	42307433392936837192597924432645724879802813461950306 06
b	=	37607854266521542731578647872763762949550357662311283 28
G	=	(25172511827447873915652640395343290609520930113828563 32, 37387948000620380485366514490443532795737683305031680 07)

p	=	66800571721904603464904980873850643323914748226599111538821622388 09
$\lfloor \log_2 p \rfloor + 1$	=	222
r	=	16700142930476150866226245218462659105547858191392594980229639835 29
$\lfloor \log_2 r \rfloor + 1$	=	220
k	=	4
a	=	66800571721904603464904980873850643323914748226599111538821622388 06
b	=	27759646820229490299377192213480107491476804804133912908160327827 75
G	=	(32730871495878678302175400076726735295945418881011746504836274682 50, 36325356287691259074699486504957001271621001476078890677663276039 83)

p	=	5331369449787147210725799182247363710162558951080875268627741810017\ 386746493149
$\lfloor \log_2 p \rfloor + 1$	=	262
r	=	1332842362446786802681449795561840927540869571121424054459187605073\ 720511411159
$\lfloor \log_2 r \rfloor + 1$	=	260
k	=	4
a	=	5331369449787147210725799182247363710162558951080875268627741810017\ 386746493146
b	=	8808842535654359666556380433494756852865478556389536949350714054793\ 98028333100
G	=	(5081731805997595463647968776224307067547060629202974902735739659 92\ 8355409323038, 8291339351528036362764097908361461263815578639057850151667575229518\ 32251490088)

p	$=$	4744331764675234892260237708941773023509425038059772525624154161285\ 80125652370918660109194 9
$\lfloor \log_2 p \rfloor + 1$	$=$	302
r	$=$	1186082941168808723065059427235443255877356259784206533290418021529\ 27946021235552136878675 9
$\lfloor \log_2 r \rfloor + 1$	$=$	300
k	$=$	4
a	$=$	4744331764675234892260237708941773023509425038059772525624154161285\ 80125652370918660109194 6
b	$=$	4437583652749548634062869480562004918339860381936577407513536467549\ 33039098867836019026865 2
G	$=$	(1212171087472766369049928283501462240320225940413610539872599687 53\ 24297974113160852459246 6, 8597985220238033439830624169219692841215495374553825267084530302901\ 51908844972867411707220)

p	$=$	7153338370075270348096695325057864552493004810586185863048089808890\ 53655293774189432675590522103547486 9
$\lfloor \log_2 p \rfloor + 1$	$=$	342
r	$=$	1788334592518817587024173831264466138123251202646547037468659034538\ 64318658800209151827775649509346941 1
$\lfloor \log_2 r \rfloor + 1$	$=$	340
k	$=$	4
a	$=$	7153338370075270348096695325057864552493004810586185863048089808890\ 53655293774189432675590522103547486 6
b	$=$	1030377414181471437025928015831066585710185924778001472839179098921\ 41115817661157505851865893816137080 6
G	$=$	(6026980849356277259329118012171052814480451881444597833598447672 82\ 10022194535266296175664298194869666 88, 5646236367291429026526062685398524078338359500178459546433232611128\ 25258949200619498253037695204529554 7)

p	$=$	6842023284346544466028236541767432094119081341362555840793762583609\ 03802782388712895305928868012034525933339848114 9
$\lfloor \log_2 p \rfloor + 1$	$=$	382
r	$=$	1710505821086636116507059135441858023529770335340638960199462892707\ 16008201138827062209244921081726133440290827899 1
$\lfloor \log_2 r \rfloor + 1$	$=$	380
k	$=$	4
a	$=$	6842023284346544466028236541767432094119081341362555840793762583609\ 03802782388712895305928868012034525933339848114 6
b	$=$	4885120389420065125503704332483277942395206457636335431347368727770\ 53520922314380402577749480747602841630469655759 9
G	$=$	(3874275032745527214752527255971935373172808828335634599165131435 86\ 73117811881845295729236104706463799190134435319 31, 1298052735167747719640880085350631298712805683248779591683024812663\ 62721989653430447288294831726928549052235262185 3)

p	$=$	713388262243919007955540009866581411393763356403952597565763029558587221325172211467773687594058379661665132588560745885216052 9
$\lfloor \log_2 p \rfloor + 1$	$=$	422
r	$=$	178347065560979751988885002466645352848440839100988149391440757353244262438693710954165464635139183548146549235455106541692532 1
$\lfloor \log_2 r \rfloor + 1$	$=$	420
k	$=$	4
a	$=$	713388262243919007955540009866581411393763356403952597565763029558587221325172211467773687594058379661665132588560745885216052 6
b	$=$	202202984617631018913971624475397765622029991012394735857701977632748229422920726760569092292600325372123681303985355670417691 0
G	$=$	(670848443168778883114597638151820757481201961884295396849717606691 9293634111186533196857522922566653905099490531858440343304336, 400517803188575530104455954200528649324297898261238489697508673673456220999221909577412369713519345745601502355489769873992189)

p	$=$	985236633254174504359488528884305568460243012081335993714292647139032071508096491290762829016048405317512789351719462055590270253756612058 9
$\lfloor \log_2 p \rfloor + 1$	$=$	462
r	$=$	246309158313543626089872132221076392115060753020333998428573161784758168756115562624015652339379085174815479860269144649935595854617356723 9
$\lfloor \log_2 r \rfloor + 1$	$=$	460
k	$=$	4
a	$=$	985236633254174504359488528884305568460243012081335993714292647139032071508096491290762829016048405317512789351719462055590270253756612058 6
b	$=$	893582171432849816681211163120703423556990320895748943829304068729170443500359895697226686242924294258462343573893379437454328068861910453 3
G	$=$	(299995370999312176864068802359171131908902103371800805114910495056012856415867165872944186783116623952965933748856323074003491163895083423 7, 744038213987659625127844083179896592789597407473484370844701461223131024506284181812261051121883939628190303713179571685974645858385483790 2)

p	$=$	732554050285729218197987463823552574916174653706241543602567995527132306084005610484510744124594067984688816340830625898845483107536827217030140818188 9
$\lfloor \log_2 p \rfloor + 1$	$=$	502
r	$=$	183138512571432304549496865955888143729043663426560385900641998881783076521052236554631922229256109984829057713859723714671036363962147261961593700158 9
$\lfloor \log_2 r \rfloor + 1$	$=$	500
k	$=$	4
a	$=$	732554050285729218197987463823552574916174653706241543602567995527132306084005610484510744124594067984688816340830625898845483107536827217030140818188 6
b	$=$	478054101180881136480316441668283602393688316112940430173483589551812750665097410174076099213205512725959608043617431288067714631828393362963347414577 9
G	$=$	(149813988095774374795669682273522551001633106803859686370548653107854220001711062433218096223550294089398085193592286649825217184347474438471069989239248680151221866832039028390366313879070799457675419806303209790425056145741136210156067225410056135095284936363975959963585939518401044133006277922188 95)

B.2.2 Elliptic Curves with $\Delta = -96599$

We present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is a prime of bitlength b in $[182; 502]$, $b \equiv 22 \bmod 40$. All curves have a cofactor k equal to 4. In addition, either group order is divisible by a prime r of bitlength $b - 2$. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is $\Delta = -96599$. Its class number is 500.

p	=	607737761809559722973679706340674619210060479536372874 9
$\lfloor \log_2 p \rfloor + 1$	=	182
r	=	151934440452389930743419926539215400590511788878961835 9
$\lfloor \log_2 r \rfloor + 1$	=	180
k	=	4
a	=	607737761809559722973679706340674619210060479536372874 6
b	=	421814595433377769199350762718490089797050451103846992 4
G	=	(856296978109005641599232995162096981527081841628097771, 165169054147102814529221182339591173319790677522240625 5)

p	=	439986544642969323036107974266750309249145515312671464152819354400 9
$\lfloor \log_2 p \rfloor + 1$	=	222
r	=	109996636160742330759026993566687478137641137119705247507874086970 1
$\lfloor \log_2 r \rfloor + 1$	=	220
k	=	4
a	=	439986544642969323036107974266750309249145515312671464152819354400 6
b	=	114224263233845554927161627162762665880807576967311565343214586325 7
G	=	(293571683864145697917397052326815155792718788181964368807450089213 3, 163846963071714555202887199793635676129278301855751313592120598550)

p	=	535625315867601401214460683872018960193485967792946436408908033918 2\ 077732828149
$\lfloor \log_2 p \rfloor + 1$	=	262
r	=	133906328966900350303615170968004740048414590927410972625306165571 6\ 315742884659
$\lfloor \log_2 r \rfloor + 1$	=	260
k	=	4
a	=	535625315867601401214460683872018960193485967792946436408908033918 2\ 077732828146
b	=	954198223656835123036673889263860907870063688803884526029973839021 9\ 07715846922
G	=	(195089780182489593301777603736059525185980674335251813250907884372\ 367143000292, 264335404697768307895634026492342777266870974772535603096316198249 3\ 339781386217)

p	=	5331480953298331065124894266452488023018648532168316706017066987712\ 8130917973726687746912 29
$\lfloor \log_2 p \rfloor + 1$	=	302
r	=	1332870238324582766281223566613122005754662133375365261849910060158\ 97185229096 1105133527119
$\lfloor \log_2 r \rfloor + 1$	=	300
k	=	4
a	=	5331480953298331065124894266452488023018648532168316706017066987712\ 8130917973726687746912 26
b	=	3362000851390284945153099035694493640836756299179807003789190557741\ 3126616978626212947580 72
G	=	(1868534183427901941180485333747094368122458723246946913783271874 74\ 0847782095323735381993892, 2844797988396564556123101404470586938206919922758319429787446084 66\ 4436486269647065104036543)

p	=	6873502563776013943542225046304599151226223661142692429065349000916\ 3563893096838825734927 11380952289489
$\lfloor \log_2 p \rfloor + 1$	=	342
r	=	1718375640944003485885556261576149787806555915285671816487218942898\ 1229552177384399535239918492854528 89
$\lfloor \log_2 r \rfloor + 1$	=	340
k	=	4
a	=	6873502563776013943542225046304599151226223661142692429065349000916\ 3563893096838825734927 11380952289486
b	=	6448784449284641098459572720420008506605013370077233640812511139368\ 8029020100970029707476828589771771531
G	=	(6277133472146921311219735713584516443195584535193250925347331745 35\ 9207318135814289203819884540172719125, 3133168889493184144323879559604245994081992122888532176029789488517\ 3215979528564537019761494461046114 10)

p	=	7627062722108113794370165995397518964237487869867693585734776811250\ 3655756216024428061875608047116485347693859300 01
$\lfloor \log_2 p \rfloor + 1$	=	382
r	=	1906765680527028448592541498849379741059371967466923396432867787027\ 3756368427046973099165414172390785811456590219 01
$\lfloor \log_2 r \rfloor + 1$	=	380
k	=	4
a	=	7627062722108113794370165995397518964237487869867693585734776811250\ 3655756216024428061875608047116485347693859299 98
b	=	5366498165998709682646604050589041277233627022582278841088353789989\ 7919250269842238623571830067955287240438469558 77
G	=	(6128597405531333447652948492714591264755058557689005565046152904 18\ 4144564601373276512330239651510343452778355467144, 4388279219896971569787997777788146681446080609652997552379630357833\ 8497691304302930348815304465937772665032372742 92)

p	$=$	10317479670848459856076341245005410741699137921210413810344379306967009742433300210659962841213888704515056154161418659658353289
$\lfloor \log_2 p \rfloor + 1$	$=$	422
r	$=$	25793699177121149640190853112513526854247844803026034525860948273936671313441689589959981564780773475208607789817626012481980 81
$\lfloor \log_2 r \rfloor + 1$	$=$	420
k	$=$	4
a	$=$	10317479670848459856076341245005410741699137921210413810344379306967009742433300210659962841213888704515056154161418659658353286
b	$=$	51853692319486045125132157080930194198390769640635714741014302354404920017586670564707669314828865308991480234606425041840199 29
G	$=$	(25841825926874642013436560026759741596900541612281149909874676342409724181867761048765243618309752713894434946803451256798053354, 15606263282604826470166447987853103845417044701751355422688474570091826786309911408145034182892877728518841453641575588439964444)

p	$=$	94651897206665936075567788028313455726629143443077160402369135009512944634502037238352565004648048002478888221282890012719659793950902 22061
$\lfloor \log_2 p \rfloor + 1$	$=$	462
r	$=$	23662974301666484018891947007078363931657285860769290100592283752378239956529762186976282634472037909478703638179711446499346599762189 89031
$\lfloor \log_2 r \rfloor + 1$	$=$	460
k	$=$	4
a	$=$	94651897206665936075567788028313455726629143443077160402369135009512944634502037238352565004648048002478888221282890012719659793950902 22058
b	$=$	38704337092462695752488842606633972320792785233670599061326066564594824052306875946439116669354366565325410212991454086976569771026151 28712
G	$=$	(69603967693345149608199911320369103761241566505634692708049610266681135325691874678322143199066493444992923089666914963470520976471199937 16, 54800228977835630574160202259630762405344344875630100893840211320128128333557557420555974725353284776972307041820625507247133357070633 76186)

p	$=$	12876786810474252623868728202870164390331438300603334543505545354879527438047451323919801656267620309902502387152360086924749226375594 479279088857110841
$\lfloor \log_2 p \rfloor + 1$	$=$	502
r	$=$	32191967026185631559671820507175410975828595751508336358763863387198818595113865478288083008277830776290703084109682123518062807313155 64609209395005821
$\lfloor \log_2 r \rfloor + 1$	$=$	500
k	$=$	4
a	$=$	12876786810474252623868728202870164390331438300603334543505545354879527438047451323919801656267620309902502387152360086924749226375594 479279088857110838
b	$=$	46083637545155568251671083016851607241077720952422867453321626642921415224929556271141447401042058326189855995022521153065713586286185 69340905293961658
G	$=$	(64536907491416739965979215860991283293137995436471262947868625602346527587612199035128378060640403400360792240356911597534544119627619268587110388906683723250448943540168571143774680406168725209235364385019670516044666018605713959255210484365827505466454602558262948771459693529232245711433156175910 05)

B.3 Elliptic Curves of Prime Order

B.3.1 Elliptic Curves over a 160-bit Field

We present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is of bitlength 160. All curves are of prime order. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is congruent 5 modulo 8. The class numbers are in $[200; 1000]$. Either discriminant is maximal and fundamental of a given class number.

Δ	=	-125579
$h(\Delta)$	=	200
p	=	831184991995438527046820150461113181889751761329
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	831184991995438527046821312318038356568772986059
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	831184991995438527046820150461113181889751761326
b	=	117759898201510996297167902939879268929396610982
G	=	(710211282216605132908873764007346339500982938581, 618437687637132233166137735257102408245679062707)

Δ	=	-184091
$h(\Delta)$	=	250
p	=	815900031752626540650354198833917562751681213199
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	815900031752626540650354701528337030533309119561
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	815900031752626540650354198833917562751681213196
b	=	700970916984443021963158115057520900520285074 42
G	=	(418723928080572569986689031558916852614983482526, 332911933974190962962949796020195494009022217811)

Δ	=	-223739
$h(\Delta)$	=	300
p	=	1034984595870937275049311029781681530862228084979
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	1034984595870937275049311420955271049089100105751
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	1034984595870937275049311029781681530862228084976
b	=	725569714846833027827650094695838128627874006402
G	=	(964182528103914974494271992150032650764137673163, 950031612704958088716295809248579652067514150 7)

Δ	=	-294971
$h(\Delta)$	=	350
p	=	1309269195941077297231344765707434896583909561801
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	1309269195941077297231345781208802920152499961879
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	1309269195941077297231344765707434896583909561798
b	=	434391801919780072135634220121662732733748945333
G	=	(581793252971854690950081335963053887603619984480, 202424508840409172379121796228404020566412608751)

Δ	=	-428819
$h(\Delta)$	=	400
p	=	1227215444871476685771634869084769115422235359141
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	1227215444871476685771635916267591800263113972109
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	1227215444871476685771634869084769115422235359138
b	=	817724535572897082170195481440459548404485112915
G	=	(583575304032094748222919431593623717042002996685, 216382461218147397617633876515550889320659329784)

Δ	=	-539579
$h(\Delta)$	=	450
p	=	10149335527097559579359906410442119131921822826 39
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	10149335527097559579359908271811356317583645182 99
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	10149335527097559579359906410442119131921822826 36
b	=	125513560580825589628491625977446558179608424065
G	=	(396147047317039466845596890767587458354743195768, 788773537721649551634338670150254419256404873704)

Δ	=	-742979
$h(\Delta)$	=	500
p	=	883110220759529382266411248915814137158154559659
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	883110220759529382266411540829503751426802665529
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	883110220759529382266411248915814137158154559656
b	=	17979474235466349842019373874962787567520864140 7
G	=	(300363866870560899854708653513151925746231107893, 475429819934886234513274746426088116805547586034)

Δ	=	-834539
$h(\Delta)$	=	600
p	=	985152662523364125375258163426313272896593404849
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	985152662523364125375259274557248364745600841889
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	985152662523364125375258163426313272896593404846
b	=	75912086401800577742166421102128389500992462910 0
G	=	(320383311988387616932517525257197671804802801153, 725256858363800960119120664802358198787743175245)

Δ	=	-1166051
$h(\Delta)$	=	700
p	=	10319787795003102237945794262287841741645752433 19
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	10319787795003102237945802226385350974800972295 69
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	10319787795003102237945794262287841741645752433 16
b	=	98997165114705531704724368070904302337793884875 4
G	=	(124700133338333671163549222169247858405401347 0, 775122973701471159313520293948941043832882577278)

Δ	=	-1390091
$h(\Delta)$	=	800
p	=	131943321536797714050581011885790838027843614302 9
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	131943321536797714050581045525355924637014120265 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	131943321536797714050581011885790838027843614302 6
b	=	396232474484060282340575670682638259187379390 01
G	=	(880136673608949488335398826948820009405778964896, 117036785658421356111708621205659464871475310988 4)

Δ	=	-1908539
$h(\Delta)$	=	900
p	=	884770987135482718657255137635954308023831616201
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	884770987135482718657255283642185738054817498029
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	884770987135482718657255137635954308023831616198
b	=	759715018180429818802132698743020337488307920962
G	=	(337864187644724783510354757013108645586795167797, 745129137363059263238813077177404356022434673572)

Δ	=	-2656979
$h(\Delta)$	=	1000
p	=	132115433571592138584439615822053157845420121602 1
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	132115433571592138584439669184023863748036881106 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	132115433571592138584439615822053157845420121601 8
b	=	754983587592374527589557853265053239553577728695
G	=	(550728092787084366574077260614392332948395261 82, 173196936949517196129783456416058826636584802124)

B.3.2 Elliptic Curves over Fields of Different Bitlength

We present elliptic curves defined over a finite prime field $\mathbb{F}_p$, where p is a prime of bitlength b in $[180; 500]$, $b \equiv 20 \bmod 40$. All curves are of prime order. The parameters of each curve are of the form $(-3, b)$ for some $b \in \mathbb{F}_p$. Furthermore, all curves respect the requirements of Section 2.3.5.

The discriminant Δ of the endomorphism ring of each curve is $\Delta = -125579$ Its class number is 200.

p	=	1321954682381448433654350458233279321390542430721856409
$\lfloor \log_2 p \rfloor + 1$	=	180
r	=	1321954682381448433654350459786766646842828378818229291
$\lfloor \log_2 r \rfloor + 1$	=	180
k	=	1
a	=	1321954682381448433654350458233279321390542430721856406
b	=	980325259892152119704529023581357799844979981845455866
G	=	(855163031696488631301639043321918738448472708955940903, 265870058284021339020750943319639320671874671748043494)

p	=	97956909246191595914223768256458927177219798799662585474446692909 9
$\lfloor \log_2 p \rfloor + 1$	=	220
r	=	97956909246191595914223768256459047413324810988749039584966254351 1
$\lfloor \log_2 r \rfloor + 1$	=	220
k	=	1
a	=	97956909246191595914223768256458927177219798799662585474446692909 6
b	=	79659450525156442140336465569000820520995352900228418986709639123 7
G	=	(635826678534782203117197325820033459838842894259450888268137376000, 397096217776286822955931690292859459241485034846840324998715975199)

p	=	131427004897150773399593618972348218666960936115860835858019378393 7\ 874472842041
$\lfloor \log_2 p \rfloor + 1$	=	260
r	=	131427004897150773399593618972348218667175598490198414927172628219 4\ 903313140959
$\lfloor \log_2 r \rfloor + 1$	=	260
k	=	1
a	=	131427004897150773399593618972348218666960936115860835858019378393 7\ 874472842038
b	=	130941842120844704815546489387407423348124485814560275308763578131 6\ 164628313882
G	=	(219915967879367026623827324824790383884843183403628793561977776404\ 902038775321, 833121702396006916914675948590500927109785441872347409279534036508 6\ 06464270757)

p	$=$	172664666893881942096688047966213509539587575245142225067052931437\3\ 9445700606526099361969 29
$\lfloor \log_2 p \rfloor + 1$	$=$	300
r	$=$	172664666893881942096688047966213509539587575324742833441259250566\6\ 5727163996374992725303 51
$\lfloor \log_2 r \rfloor + 1$	$=$	300
k	$=$	1
a	$=$	172664666893881942096688047966213509539587575245142225067052931437\3\ 9445700606526099361969 26
b	$=$	757223383900348694714065931901590230735205487197650653216604581443\3\ 9599067874053606234376 6
G	$=$	(132237344740368492427294346980686989813412907129655609203384255722\\ 41673239864637254046541 77, 818459195056854702139636350143170416182640178833041208944357642452 9\\ 6810576499277962684142 8)

p	$=$	130465222832370138812838152451584258142342598666215381177078641779 1\ 7234634019524273837851428325891537 31
$\lfloor \log_2 p \rfloor + 1$	$=$	340
r	$=$	130465222832370138812838152451584258142342598666215507506844005185 8\ 1710007784546415196675248436985998 39
$\lfloor \log_2 r \rfloor + 1$	$=$	340
k	$=$	1
a	$=$	130465222832370138812838152451584258142342598666215381177078641779 1\ 7234634019524273837851428325891537 28
b	$=$	702401062081564658951534880841668448915115733181475650572555517483 9\ 7910336004356585538352352132233 34
G	$=$	(166470852591772535515977977449796814905608282128022144767524765067\\ 73409316472111786860653621481471 1924, 539366273369031547159705743302230777151584945420106080599949754573 1\\ 6774669843759561874438854618703891 1)

p	$=$	210768393425478615357860718259234995798967010348376774084589557911 9\ 8577919577908597772580920247855665612485445000 59
$\lfloor \log_2 p \rfloor + 1$	$=$	380
r	$=$	210768393425478615357860718259234995798967010348376774084710826558 7\ 0753259481859618351935576628912958264206232858 41
$\lfloor \log_2 r \rfloor + 1$	$=$	380
k	$=$	1
a	$=$	210768393425478615357860718259234995798967010348376774084589557911 9\ 8577919577908597772580920247855665612485445000 56
b	$=$	378452313910859513793578643985762475610361290460898372404099983537 6\ 4544090661394678651679044018191298985647370 5673
G	$=$	(286085962156093267621914086536217547556130851511119826934580761949\\ 80558689261446022936359205250032246698039 2677, 181607539081716565531828761063602990811934406942249674317360328933 6\\ 3544050814204202831650481715477960859132071868 80)

p	$=$	21053814540876485733749686936213664035197543825927977633322606777 37\ 87195426879386077857584110386308109570957388444106660127 5019
$\lfloor \log_2 p \rfloor + 1$	$=$	420
r	$=$	21053814540876485733749686936213664035197543825927977633322606785 72\ 59278542449955515694153037725274567443973595747335927436857 1
$\lfloor \log_2 r \rfloor + 1$	$=$	420
k	$=$	1
a	$=$	21053814540876485733749686936213664035197543825927977633322606777 37\ 87195426879386077857584110386308109570957388444106660127 5016
b	$=$	19737833953195850581531514851625274403045654404558616395468801181 26\ 40471773774187993781559772793712580783929870309026712149118
G	$=$	(18090152549741002743435953481383888077438414570346054759661637536 0\ 49201943047122310048761461714931984083708248478453543231319 51, 16389635230328520017524432773947627655085501376675366469345564700 55\ 82786175621134128487242445168027307982126714430063215841779 8)

p	$=$	19927263923822367466274200070884694368552514627303833785101724768 69\ 39434858799153798126503921604880861552474265316928803524504063082 80\ 16331
$\lfloor \log_2 p \rfloor + 1$	$=$	460
r	$=$	19927263923822367466274200070884694368552514627303833785101724768 69\ 39658054575857633523103896112832577938401337290681558312401954609 18\ 92839
$\lfloor \log_2 r \rfloor + 1$	$=$	460
k	$=$	1
a	$=$	19927263923822367466274200070884694368552514627303833785101724768 69\ 39434858799153798126503921604880861552474265316928803524504063082 80\ 16328
b	$=$	53141360438415027133482447818386018754897788426343991313999239371 20\ 80629713715370755694051560680346700222290458390612701069555561612 47\ 8930
G	$=$	(69436271135287861228057300663608419197881722295326672676203378174 7\ 97253313210967586318983058643954763075909848458068617873093431652 94\ 31850, 66302696709470806535999994218120123760640657099594457911677458930 73\ 38485470145335912825905501003650609746301485775093221427387898652 68\ 4874)

p	$=$	22740925575766036808911277042475742094471545226597283377015832192 15\ 31083226933521505133011926150560844119233630742837391775196641686 82\ 43576268395457649
$\lfloor \log_2 p \rfloor + 1$	$=$	500
r	$=$	22740925575766036808911277042475742094471545226597283377015832192 15\ 31083227116108503169655139409768087808032478653620713451107156163 95\ 23472249705845639
$\lfloor \log_2 r \rfloor + 1$	$=$	500
k	$=$	1
a	$=$	22740925575766036808911277042475742094471545226597283377015832192 15\ 31083226933521505133011926150560844119233630742837391775196641686 82\ 43576268395457646
b	$=$	28684074068793351303631041187096822941024995743518654806131037299 85\ 48080699469501926113906526638787147366097840270001718300710433611 79\ 9513278451418660
G	$=$	(77837159650878610123247493434365328451080286916812832114075007486 3\ 17993753351176199576848125199581089175928050964863974980942646402 04\ 4253406975821355, 22454405460850205475947654700197348545982352501027380272726899464 74\ 48830628053562835837712364935010116242997072800948941079098886135 64\ 39137548736922034)

B.4 Sample Twisted Pairs

B.4.1 Twisted Pairs over a 160-bit field

We list some sample twisted pairs defined over a finite prime field $\mathbb{F}_p$, where p is of bitlength 160. According to Definition 11.2.1 all elliptic curves and their twists are of prime order and respect the requirements of Section 2.3.5, respectively. The parameters of each curve and its twist are of the form $(-3, b)$ and $(-3, b')$, respectively, for some $b, b' \in \mathbb{F}_p$. As explained in Section 11.2, we have to ensure $p \equiv 3 \bmod 4$.

The discriminant Δ of the endomorphism ring of each curve is congruent 5 modulo 24. The class numbers are in $[200; 500]$. Either discriminant Δ is maximal and fundamental of a given class number with the property $\Delta \equiv 5 \bmod 24$.

Δ	=	-356131
$h(\Delta)$	=	200
p	=	859930367154478455706777652934856251164297185639
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	859930367154478455706776773256618162702869307881
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	859930367154478455706777652934856251164297185636
b	=	528228061876465954195650594528318719872952412079
G	=	(855442175319922008831959249774161110500747605373, 612536680028626994386544817765470437702610581287)
r'	=	859930367154478455706778532613094339625725063399
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	859930367154478455706777652934856251164297185636
b'	=	331702305278012501511127058406537531291344773560
G'	=	(142835027876730419442745454782072148193643873423, 679922187991150908578451755494872540938925191119)

Δ	=	-467011
$h(\Delta)$	=	250
p	=	1446495879078530457450264992260584792175973538359
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	1446495879078530457450263185220247294365002286329
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	1446495879078530457450264992260584792175973538356
b	=	1183048817443472037120941710534367126849506482622
G	=	(442723487819934872992539753898522906124104552121, 109346706582444365665742134326715555038686412871 6)
r'	=	1446495879078530457450266799300922289986944790391
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	1446495879078530457450264992260584792175973538356
b'	=	263447061635058420329323281726217665326467055737
G'	=	(118296783013122205681124015008668947804253633637 1, 883708187899748063485537949411444981588689134529)

Δ	=	-881011
$h(\Delta)$	=	300
p	=	139579622397262943207528143838090271717083702755 9
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	139579622397262943207528055332884633549462114487 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	139579622397262943207528143838090271717083702755 6
b	=	50260249199104064754472372153770943898267890181 9
G	=	(21774007268509865390051090749202573267204136763 2, 51082903529503215588401700136673764403441079736)
r'	=	139579622397262943207528232343295909884705291024 1
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	139579622397262943207528143838090271717083702755 6
b'	=	89319373198158878453055771684319327818815812574 0
G'	=	(13592030031404991680011655934988540847625757560 21, 16723344335322068696991968294229053968450629503 1)

Δ	=	-1190251
$h(\Delta)$	=	350
p	=	123013399579986384679543160293427336803903705289 9
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	123013399579986384679542955322935485298193183803 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	123013399579986384679543160293427336803903705289 6
b	=	36992019576738665059575492224669302853829870735 5
G	=	(45656890451837258947471292446983383100045297034 7, 44815667234491488470593715207417940735899576747 0)
r'	=	123013399579986384679543365263919188309614226776 1
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	123013399579986384679543160293427336803903705289 6
b'	=	86021380003247719619967668068758033950073834554 4
G'	=	(88174951803229711286066244059622305185468193854 6, 81620567881093271068462886425625297926533611161)

Δ	=	-1531339
$h(\Delta)$	=	400
p	=	123380439304535093052380349255743049290322179405 9
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	123380439304535093052380199099215312335675294712 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	123380439304535093052380349255743049290322179405 6
b	=	88864065882265364075842768316290871122312405525 6
G	=	(61283033056113821190054631608597273095302709421 8, 10469169085845947574340631769066985221605606934 63)
r'	=	123380439304535093052380499412270786244969064099 1
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	123380439304535093052380349255743049290322179405 6
b'	=	34516373422269728976537580939452178168009773880 3
G'	=	(16449488808715488035874959867806977151059699812 4, 20895199146125342926320282749728489100957729585)

Δ	=	-1825291
$h(\Delta)$	=	450
p	=	74348684156960967853139015293497080380147308811 9
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	74348684156960967853138983869884840358706127586 9
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	74348684156960967853139015293497080380147308811 6
b	=	68663438057328038394966389929088336891224453256 7
G	=	(31494713903663406575656083992816740234851685124 8,
		23254548100832641795105873518532099209603914677)
r'	=	74348684156960967853139046717109320401588490037 1
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	74348684156960967853139015293497080380147308811 6
b'	=	56852460996329294581726253644087434889228555552
G'	=	(43326655266986525963435800844349629348824155537 5,
		30629635116055538850936220874601060083882152129 0)

Δ	=	-2299699
$h(\Delta)$	=	500
p	=	12665947870974942849656880333475615414124205235 39
$\lfloor \log_2 p \rfloor + 1$	=	160
r	=	12665947870974942849656870454830361471647520433 31
$\lfloor \log_2 r \rfloor + 1$	=	160
k	=	1
a	=	12665947870974942849656880333475615414124205235 36
b	=	26900032018036742758751195345438439950531216098 0
G	=	(51874706663255022463998931231761060743117323021 7,
		11541996551340646385744254373245590457320797232 26)
r'	=	12665947870974942849656890212120869356600890037 49
$\lfloor \log_2 r' \rfloor + 1$	=	160
k'	=	1
a'	=	12665947870974942849656880333475615414124205235 36
b'	=	99759446691712685737817607989317714190710836255 9
G'	=	(55200735905075060558315779674130247223008217641 7,
		32869990089254619289655021688600637038691736996 0)

B.4.2 Twisted Pairs over Fields of Different Bitlength

We present sample twisted pairs defined over a finite prime field $\mathbb{F}_p$, where p is a prime of bitlength b in $[180; 500]$, $b \equiv 20 \bmod 40$. The parameters of each curve and its twist are of the form $(-3, b)$ and $(-3, b')$, respectively, for some $b, b' \in \mathbb{F}_p$. All primes are congruent 3 modulo 4.

The discriminant Δ of the endomorphism ring of each curve is $\Delta = -356131$. Its class number is 200. Δ is maximal of class number 200 with $\Delta \equiv 5 \bmod 24$.

p	=	1075433050694156749250466429969802312218754032972784619
$\lfloor \log_2 p \rfloor + 1$	=	180
r	=	1075433050694156749250466428493633170987557002772535269
$\lfloor \log_2 r \rfloor + 1$	=	180
k	=	1
a	=	1075433050694156749250466429969802312218754032972784616
b	=	563697610298953819558072871221376143881612339876587799
G	=	(997918472409719873494412251770042564384396804059617448, 292863164766361847397780288047281543603146765349893515)
r'	=	1075433050694156749250466431445971453449951063173033971
$\lfloor \log_2 r' \rfloor + 1$	=	180
k'	=	1
a'	=	1075433050694156749250466429969802312218754032972784616
b'	=	511735440395202929692393558748426168337141693096196820
G'	=	(100373746085586884161242016743677599594432517381232566 3, 711187556205908079232181220096335658544592704083093803)

p	=	152902880845425342398997236781361922784864241451406017480370379 1699
$\lfloor \log_2 p \rfloor + 1$	=	220
r	=	152902880845425342398997236781361898475047206558553798868739053 8089
$\lfloor \log_2 r \rfloor + 1$	=	220
k	=	1
a	=	152902880845425342398997236781361922784864241451406017480370379 1696
b	=	100506970251789438361478814544960731955713724959711069917635491 6549
G	=	(263806037213223864985327923825153098481928836785824437211705646586, 152257752999668768202248369968447541129046562027535015566086259206 1)
r'	=	152902880845425342398997236781361947094681276344258236092001704 5311
$\lfloor \log_2 r' \rfloor + 1$	=	220
k'	=	1
a'	=	152902880845425342398997236781361922784864241451406017480370379 1696
b'	=	523959105936359040375184222364011908291505164916949475627348875150
G'	=	(487320795236796724639862588098511961107904594290512968211171227711, 121963342709578307487605901215643178201926869601734339471787064906 7)

p	$=$	1593758134703409939757852593188441169132595610102811674126250787025\ 662881712459
$\lfloor \log_2 p \rfloor + 1$	$=$	260
r	$=$	1593758134703409939757852593188441169131096807658308925517319374225\ 342118861429
$\lfloor \log_2 r \rfloor + 1$	$=$	260
k	$=$	1
a	$=$	1593758134703409939757852593188441169132595610102811674126250787025\ 662881712456
b	$=$	876564263913688507705775625629317488822291943120745421372550603788 2\ 46105465157
G	$=$	(341627581636816035282177706138590727019533551830466597068276406839\ 225703219207, 725878076562363155931974401333228903831624004232021737564147982704 0\ 79446839003)
r'	$=$	1593758134703409939757852593188441169134094412547314422735182199825\ 983644563491
$\lfloor \log_2 r' \rfloor + 1$	$=$	260
k'	$=$	1
a'	$=$	1593758134703409939757852593188441169132595610102811674126250787025\ 662881712456
b'	$=$	7171938707897214320520769675591236803103036669820662527537001832374\ 16776247302
G'	$=$	(9510584004949009825926353332329887199590216515955279798295120926 40\ 981294612084, 161717361671200291132793257020187887483614970851914398021595783918 4\ 90242000293)

p	$=$	1275659457698428492563982286720172061188626264206879037707206800324\ 67952777463156527939469 9
$\lfloor \log_2 p \rfloor + 1$	$=$	300
r	$=$	1275659457698428492563982286720172061188626263931438050832866850383\ 60360850922767963450833 9
$\lfloor \log_2 r \rfloor + 1$	$=$	300
k	$=$	1
a	$=$	1275659457698428492563982286720172061188626264206879037707206800324\ 67952777463156527939469 6
b	$=$	1158158716004525703574266480912701194410624068852266688964413270537\ 73241157297177327825985 8
G	$=$	(592766759477268955204109519454055387685102212923266598157528533322\ 90573726719359560921193 3, 113432438840542519166311681603582909419410114917366206672998947568 1\ 49995374295216615107474 4)
r'	$=$	1275659457698428492563982286720172061188626264482320024581546750265\ 75544704003545092428106 1
$\lfloor \log_2 r' \rfloor + 1$	$=$	300
k'	$=$	1
a'	$=$	1275659457698428492563982286720172061188626264206879037707206800324\ 67952777463156527939469 6
b'	$=$	1175007416939027889897158058074708667780021953546123487427935297869\ 47116201659792001134841
G'	$=$	(764263285083135331836012434315591298215236564988200630598240338974\ 99710071843458017482111, 659128669451920411632818929358666733937450536252192947571253352457 4\ 98652668255660477840899)

p	$=$	193136786740013321422451800679446606191739783423261632482136636223899706409724328259388497335135930769
$\lfloor \log_2 p \rfloor + 1$	$=$	340
r	$=$	193136786740013321422451800679446606191739783423261417782523962443936676615649871731143868689944222106
$\lfloor \log_2 r \rfloor + 1$	$=$	340
k	$=$	1
a	$=$	193136786740013321422451800679446606191739783423261632482136636223899706409724328259388497335135930769
b	$=$	10002068256985710971588982058531067495924709383739119303998077701710483827804194273253318789778879007
G	$=$	(76236880712025722467949525201352740856503450166218156362409305064190851091740222557199517166342377378, 12741550953689636386114612577748948975888656548813024593161703481995708658288924998289133848506811128997)
r'	$=$	193136786740013321422451800679446606191739783423261847181749310003862736203798784787633125980327639433 9
$\lfloor \log_2 r' \rfloor + 1$	$=$	340
k'	$=$	1
a'	$=$	193136786740013321422451800679446606191739783423261632482136636223899706409724328259388497335135930769
b'	$=$	93116104170156211706561980094135931232492689585870439442155859206794868131682385526855309437347140673 2
G'	$=$	(18566660852630882538593762055522412560094823055806890024935995481944530677503790534488086417153313209 40, 74365614431980375234985616654767828516330457559270597397104334652780607593236072874777312730187053252 7)

p	$=$	19745772902159217524741798460670908856429467462445476543878270951219564947080965249474219197169001509492196349100 59
$\lfloor \log_2 p \rfloor + 1$	$=$	380
r	$=$	19745772902159217524741798460670908856429467462445476543855853316491063519523424451731850218449720673064336341073 79
$\lfloor \log_2 r \rfloor + 1$	$=$	380
k	$=$	1
a	$=$	19745772902159217524741798460670908856429467462445476543878270951219564947080965249474219197169001509492196349100 56
b	$=$	16531814356727687791768393340828074602310376883045602962455976702394170935885281536946881473337766972250410637914 38
G	$=$	(94896142832134569808610370486452076306986961500647761613756442403923738317134446902680062078369264475504086321530 8, 88854834146832560979887703090920175823703322441117047555932473659607664390000477610570228947029825430641311855292)
r'	$=$	19745772902159217524741798460670908856429467462445476543900688585948066374638506047216588175888282345920056357127 41
$\lfloor \log_2 r' \rfloor + 1$	$=$	380
k'	$=$	1
a'	$=$	19745772902159217524741798460670908856429467462445476543878270951219564947080965249474219197169001509492196349100 56
b'	$=$	32139585454315297329734051198428342541190905793998735814222942488253940111956837125273377238312345372417857111 8621
G'	$=$	(15815967454934344843609031426537728104294309065101458804673457832313885543810860307837936713429004557463105412981 52, 47374082352260470995280290008865088157985110310538329488512848757496992031982381178024588793378221448108701782653 9)

p	$=$	25998942737152728500086544450232692350970636140151449874605221260 81\ 97919979668653167733134208489871109060418226017683960179639 9
$\lfloor \log_2 p \rfloor + 1$	$=$	420
r	$=$	25998942737152728500086544450232692350970636140151449874605221234 19\ 86438604140620656662226234122511678197853743364676702810888 9
$\lfloor \log_2 r \rfloor + 1$	$=$	420
k	$=$	1
a	$=$	25998942737152728500086544450232692350970636140151449874605221260 81\ 97919979668653167733134208489871109060418226017683960179639 6
b	$=$	20332162901129207062232373062123274263657690820319315984962137614 42\ 91264845242175915062736043784554980720576760632918412101646
G	$=$	(71508137875102442628999019781196474951184125986126887101041387667 4\ 02618560122808957107722998756578228254233018873298990460517 1, 40526681178966756211129109817249098404977047258742984267477432183 62\ 33118187588977857759106103847541061740497426867055206489899)
r'	$=$	25998942737152728500086544450232692350970636140151449874605221287 44\ 09401355196685678804042182857230539922982708670691217548391 1
$\lfloor \log_2 r' \rfloor + 1$	$=$	420
k'	$=$	1
a'	$=$	25998942737152728500086544450232692350970636140151449874605221260 81\ 97919979668653167733134208489871109060418226017683960179639 6
b'	$=$	23965726447039807793863307144020364924604867058119518276109007499 37\ 68793495144435576226860604111415610988360549954392118969475 3
G'	$=$	(24416891544992996953020793432708318187842161525816271480339775326 5\ 68562435298745788913729779100214318168180201126550619168914 17, 15453739832217839359356432520950047947577415414656376652401263807 53\ 60996754712968101762241262963748648743117787609790168155632 1)

p	$=$	19577115299328387909042917759476695313505044182221147495872911257 81\ 44572432019466180345756102732799985725916316805238855519257389359 93\ 41319
$\lfloor \log_2 p \rfloor + 1$	$=$	460
r	$=$	19577115299328387909042917759476695313505044182221147495872911257 81\ 44556215781502942718766027507765007273270872653154674864791223650 69\ 69569
$\lfloor \log_2 r \rfloor + 1$	$=$	460
k	$=$	1
a	$=$	19577115299328387909042917759476695313505044182221147495872911257 81\ 44572432019466180345756102732799985725916316805238855519257389359 93\ 41316
b	$=$	38132180341631633413530102261094508480429824894304164772364018022 01\ 30373889788093005708058204063202662493304501596376746146215544321 53\ 5060
G	$=$	(12085142042451554118707711056567475986412357792407748262812327983 8\ 60338217911018335246111739350261860866787321769042698800290752564 54\ 735103, 53848475999885356347853787290471421999915568278693379281735290595 85\ 81976709743046053952643120567178831875737593524663782020213912323 84\ 6904)
r'	$=$	19577115299328387909042917759476695313505044182221147495872911257 81\ 44588648257429417972746177957834964178561760957323036173723555069 17\ 13071
$\lfloor \log_2 r' \rfloor + 1$	$=$	460
k'	$=$	1
a'	$=$	19577115299328387909042917759476695313505044182221147495872911257 81\ 44572432019466180345756102732799985725916316805238855519257389359 93\ 41316
b'	$=$	15763897265165224567689907533367244465462061692790731018636509455 61\ 31535043040656879774950282326479719476585866645601180904635834927 78\ 06259
G'	$=$	(50276637261131517079959123996057706681289714509029782333881187059 6\ 15791735341224400773019529779919044112029243279239780409964286231 2\ 44291, 16513657999872993808911512384800979653498284724293563790764405695 04\ 95383697252518389258867701394812568033428012806216156202928090214 82\ 6950)

p	$=$	19546031473547635850915067098927591278568535324173110173126754951765967933182955626941001355654115920803649992475230037831218619024289\46909018727838899
$\lfloor \log_2 p \rfloor + 1$	$=$	500
r	$=$	19546031473547635850915067098927591278568535324173110173126754951765967933169244421726464188758984637293774864062424475656337812609748\87473131217570389
$\lfloor \log_2 r \rfloor + 1$	$=$	500
k	$=$	1
a	$=$	19546031473547635850915067098927591278568535324173110173126754951765967933182955626941001355654115920803649992475230037831218619024289\46909018727838896
b	$=$	10175060048331960168339169971497120918245517898677828905634172092080748919897238180597537607211945054334281014369085615079952033170085\7743664845918984
G	$=$	(17485114218967029276245830442812049267630550508332941917827089154694973139799363469218784496824287273068744923475893935121273243685089867552768876423,\57113855297206520475600778000119769677013214348657666987255415447956432919125092553604729678860398972457739661426711002889291163019357\6684096025923013)
r'	$=$	19546031473547635850915067098927591278568535324173110173126754951765967933196666832155538522549247204313525120888035600006099425438830\06344906238107411
$\lfloor \log_2 r' \rfloor + 1$	$=$	500
k'	$=$	1
a'	$=$	19546031473547635850915067098927591278568535324173110173126754951765967933182955626941001355654115920803649992475230037831218619024289\46909018727838896
b'	$=$	18528525468714439834081150101777879186743983534305327282563337742557893041193231808881247594932921415370221891038321476323223415707280\89165353881919915
G'	$=$	(97684150003858480515884351240564807381075600517953375390612076001713721870848737336426215664609298529543603286851336477337292745681029535365606721615,\14940476273893284486049505009490342173945556511676784341018261500626785006568241657373316996327904182127351704586641696592170637349211\61026502451984236)

B.5 Sample Elliptic Curves over Optimal Extension Fields

We give sample elliptic curves of prime order defined over Optimal Extension Fields suitable for use in cryptography. The curves are output of `findOEF`$(32, 5, h_0)$ for $200 \leq h_0 \leq 500$, $50 \mid h_0$.

h_0	=	200
p	=	4294920991
p^5	=	1461422855154656276375246691389613755707993345951
r	=	1461422855154656276375244308087074996229811558097
Δ	=	-125579
h	=	200
f	=	$X^5 - 2$
a	=	$4268796136X^4 + 608938058X^3 + 596605303X^2$ $+4138815490X + 4132637335 \bmod f$
b	=	$278978933X^4 + 411480498X^3 + 3077264248X^2$ $+3245827116X + 2284547224 \bmod f$
G_x	=	$1778236220X^4 + 275799952X^3 + 645659615X^2$ $+2510908108X + 3042669843 \bmod f$
G_y	=	$1939043504X^4 + 3721906723X^3 + 3329509813X^2$ $+1078338789X + 4117271912 \bmod f$

h_0	=	250
p	=	4294940641
p^5	=	1461456286761355436053754012243104344770295439201
r	=	1461456286761355436053755427217314521582314459929
Δ	=	-254579
h	=	250
f	=	$X^5 - 3$
a	=	$162530878X^4 + 1434759357X^3 + 3360802834X^2$ $+2325410416X + 1615712460 \bmod f$
b	=	$1855689095X^4 + 2040806798X^3 + 1831304219X^2$ $+3017388316X + 4105657423 \bmod f$
G_x	=	$25846706X^4 + 2626164724X^3 + 1930438558X^2$ $+3983702370X + 453012047 \bmod f$
G_y	=	$118998812X^4 + 3402773465X^3 + 2495112494X^2$ $+2606285246X + 2711221906 \bmod f$

h_0	=	300
p	=	4294946041
p^5	=	1461465474180289675134151386194236101750995386201
r	=	1461465474180289675134153762228208866377971067629
Δ	=	-414851
h	=	300
f	=	$X^5 - 2$
a	=	$3063095858X^4 + 2250048299X^3 + 1227629041X^2$ $+2778000899X + 2334717501 \bmod f$
b	=	$2011771569X^4 + 2450373594X^3 + 2713668309X^2$ $+1343344406X + 3782740697 \bmod f$
G_x	=	$2519765339X^4 + 1484568211X^3 + 1890379562X^2$ $+4033214317X + 2772079756 \bmod f$
G_y	=	$1423502591X^4 + 325230747X^3 + 1248217441X^2$ $+2075517958X + 3483474078 \bmod f$

h_0	=	350
p	=	4294922341
p^5	=	1461425151963024659833421718534534696672193607701
r	=	1461425151963024659833421799241929059403985908529
Δ	=	-814379
h	=	355
f	=	$X^5 - 2$
a	=	$1090432821X^4 + 1845057900X^3 + 3930518830X^2$ $+3187193331X + 1821748098 \bmod f$
b	=	$837115239X^4 + 3913395918X^3 + 4054795192X^2$ $+1342690143X + 2470311794 \bmod f$
G_x	=	$3659778998X^4 + 1629153926X^3 + 3716994632X^2$ $+3999793509X + 1252226858 \bmod f$
G_y	=	$1687217169X^4 + 3131806240X^3 + 2429646086X^2$ $+746907931X + 3665787462 \bmod f$

h_0	=	400
p	=	4294921051
p^5	=	1461422957234966883894402493148363918278475630251
r	=	1461422957234966883894400567650974759228682544723
Δ	=	-850043
h	=	405
f	=	$X^5 - 3$
a	=	$2503430324X^4 + 1389142557X^3 + 153775671X^2$ $+3912616042X + 1024738588 \bmod f$
b	=	$635763311X^4 + 3453827885X^3 + 1532141585X^2$ $+3160150527X + 574724140 \bmod f$
G_x	=	$4149911725X^4 + 4229024405X^3 + 595953743X^2$ $+756779967X + 971847968 \bmod f$
G_y	=	$3897363592X^4 + 4179338582X^3 + 3448789063X^2$ $+1290370184X + 597283985 \bmod f$

h_0	=	450
p	=	4294957081
p^5	=	1461484257491683831369162989777073474456006769401
r	=	1461484257491683831369163301118907834828306571529
Δ	=	-717659
h	=	455
f	=	$X^5 - 3$
a	=	$3547595190X^4 + 1549130649X^3 + 309575423X^2$ $+1026341689X + 3542034721 \bmod f$
b	=	$1891696257X^4 + 3714031196X^3 + 1643008985X^2$ $+1950309539X + 1112960195 \bmod f$
G_x	=	$2971758684X^4 + 3367594823X^3 + 3387263308X^2$ $+3036356326X + 1513878336 \bmod f$
G_y	=	$365879670X^4 + 1139499263X^3 + 326478353X^2$ $+1826093072X + 3369983575 \bmod f$

h_0	=	500
p	=	4294906591
p^5	=	1461398356045076634353290373406880319279418553951
r	=	1461398356045076634353292284980322466135601171777
Δ	=	-880739
h	=	500
f	=	$X^5 - 2$
a	=	$1025225944X^4 + 1252157679X^3 + 4290687620X^2$ $+2427519008X + 2603133748 \bmod f$
b	=	$2753540856X^4 + 2701278011X^3 + 3640333909X^2$ $+1638819043X + 786445471 \bmod f$
G_x	=	$2597365134X^4 + 1211849507X^3 + 1115371594X^2$ $+1080009322X + 344361619 \bmod f$
G_y	=	$3178621928X^4 + 3717126658X^3 + 2776153561X^2$ $+654821585X + 844138528 \bmod f$

Appendix C

The gec-manual

Name

`gec_complex_multiplication`... class for generating elliptic curves for use in cryptography

Abstract

The class `gec_complex_multiplication` generates elliptic curves over finite fields suitable for use in cryptography. It uses the theory of complex multiplication.

Description

`gec_complex_multiplication` is a class for generating elliptic curves over finite fields suitable for use in cryptography. The class implements algorithms which use the theory of complex multiplication. Currently, curves over finite prime fields and curves over Optimal Extension Fields may be generated using this package.

A central term in the theory of complex multiplication is an *imaginary quadratic discriminant*. We denote such a discriminant by Δ. An imaginary quadratic discriminant is simply a negative integer congruent 0 or 1 modulo 4. Associated to Δ is a class number, which we denote by $h(\Delta)$. In addition, for each imaginary quadratic discriminant Δ there exists a fundamental discriminant Δ_K; Δ_K is the largest imaginary quadratic discriminant dividing Δ.

Let p be a prime, and let m be either 1 or a prime. We set $q = p^m$. Let $E = (a_4, a_6)$ be an elliptic curve defined over $\mathbb{F}_q$. E is suitable for cryptographic purposes, if it respects the following conditions ([GIS01]):

1. We have $|E(\mathbb{F}_{p^m})| = r \cdot k$ with a prime $r > 2^{159}$ and a positive integer $k \leq 4$.
2. The primes r and p are different.
3. The order of q in $\mathbb{F}_r^\times$ is at least $B := 10^4$.
4. The class number of the maximal order which contains $\mathrm{End}(E)$ is at least $h_0 := 200$.

However, the choices B and h_0 may be set differently be setting the boolean `according_to_bsi` to `false`. If `according_to_bsi=false`, the last requirement is not taken into account. In addition, B has to be at least B_0 in this case, where B_0 is minimal with $r^{B_0} > 2^{1999}$.

If another security level is required, the user may invoke `set_lower_bound_bitlength_r` or `set_upper_bound_k` to set different bounds for r and k, respectively. For example, if r has to be of bitlength at least 180, that is $r > 2^{179}$, one has to invoke `set_lower_bound_bitlength_r` with 180 as argument.

During the computation the algorithm computes a class polynomial, that is a polynomial with coefficients in $\mathbb{Z}$ generating the ring class field of the imaginary quadratic order of discriminant Δ over $\mathbb{Q}(\sqrt{\Delta})$. If $3 \nmid \Delta$, we make use of a polynomial G due to Atkin/Morain to generate the ring class field. If in addition $\Delta \equiv 1 \bmod 8$, the algorithm uses a polynomial W due to Yui/Zagier. If $3 \nmid \Delta$, the class member `generation_mode` is set to 3. If in addition $\Delta \equiv 1 \bmod 8$, `generation_mode` is set to 4. Furthermore, if `generation_mode` $= 1$, then the ring class polynomial H is used.

The boolean VERBOSE may be set to `true` to get a lot of information during the computation.

Constructors/Destructor

`ct gec_complex_multiplication ()`
initializes an empty instance.

`ct gec_complex_multiplication (const bigint &` D`)`
initializes an instance and sets $\Delta = D$.

`ct gec_complex_multiplication (lidia_size_t` d`)`
initializes an instance and sets the extension degree of the finite field $\mathbb{F}_q$ over $\mathbb{F}_p$ to d.

`dt ~gec_complex_multiplication ()`

Assignments

Let I be an instance of `gec_complex_multiplication`.

`void` I`.set_field (const bigint &` l`)`
sets $p = l$.

`void` I`.set_degree (lidia_size_t` d`)`
sets degree of finite field $\mathbb{F}_q$ over its prime field to d.

Mutators of static variables to set security level.

`void` I`.set_default_lower_bound_bitlength_r (lidia_size_t` b_0`)`
sets default lower bound of bitlength of r to b_0. The default value is 160.

`void` I`.set_default_upper_bound_k (const bigint &` k_0`)`
sets default upper bound of k to k_0. The default value is 4.

`void` I`.set_default_lower_bound_extension_bitlength (lidia_size_t` l`)`
sets default lower bound of the bitlength of finite fields in which the discrete logarithm problem is considered intractable, to l. The default value is 2000.

Mutators of non-static variables to set security level.

`void` I`.set_lower_bound_bitlength_r (lidia_size_t` b_0`)`
sets lower bound of bitlength of r to b_0. The prime r will satisfy $r > 2^{b_0-1}$.

`void` I`.set_upper_bound_k (const bigint &` k_0`)`
sets upper bound of k to k_0. The cofactor k will respect $k \leq k_0$.

Mutators to set variables in the context of the requirements of the German Information Security Agency (GISA, BSI).

`void` I`.set_according_to_BSI (bool` b`)`
if b is equal to `true`, the requirements 1 - 4 cited above are respected. If b is equal to `false`, requirement 4 is ignored. In addition, in requirement 3 we only ensure that the order of q in $\mathbb{F}_r^\times$ is at least B_0, where B_0 is minimal with $r^{B_0} > 2^{b_0-1}$. b_0 is equal to `default_lower_bound_extension_bitlength`.

`void` I`.set_BSI_lower_bound_h_field (lidia_size_t` h_0`)`
ensures that the maximal imaginary quadratic order corresponding to Δ has a class number at least h_0. The default value is 200 (see requirement 4). This function is only relevant if `according_to_BSI=true`.

`void` I`.set_BSI_lower_bound_extension_degree (lidia_size_t` d_0`)`
ensures that the order of q in $\mathbb{F}_r^\times$ is at least d_0. The default value is 10000 (see requirement 3). This function is only relevant if `according_to_BSI=true`.

Mutators which are specific to `gec_complex_multiplication`.

`void` I`.assign (const gec_complex_multiplication & J)`
initializes I with J.

`void` I`.set_delta (const bigint &` D`)`
sets $\Delta = D$.

`void` I`.set_generation_mode (unsigned int` i`)`
sets `generation_mode` $= i$. The user is responsible that i and Δ fit together.

`void` I`.set_complex_precision (long` i`)`
sets the floating point precision to compute the class polynomial corresponding to `generation_mode` to i. The user is responsible that the precision is sufficient to get a correct polynomial.

`void` I`.set_class_polynomial (const polynomial<bigint> &` g`)`
sets the class polynomial to g.

`void` I`.set_class_polynomial (const bigint &` D`, const lidia_size_t` h`,`
`char * input_file)`
sets $\Delta = D$ and the class number to h. The user is responsible that $h(\Delta) = h$. Let $C = \sum_{k=0}^{h} c_k X^k$ denote a class polynomial corresponding to Δ. The coefficients are read from the file h_¡h¿/¡D¿ in the directory ¡input_file¿, beginning with c_h.

Access Methods

Let I be an instance of `gec_complex_multiplication`. Most of the accessors are only meaningful if the method `generate` has been invoked before.

Accessors for finite field, elliptic curve and its order.

`const bigint &` I`.get_p () const`
: returns p. Either p has been set using `set_field` or the method `generate` has been invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_q () const`
: returns the prime power $q = p^m$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_r () const`
: returns the prime r. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_k () const`
: returns the cofactor k. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const gf_element &` I`.get_a4 () const`
: returns the parameter a_4 of the curve $E = (a_4, a_6)$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const gf_element &` I`.get_a6 () const`
: returns the parameter a_6 of the curve $E = (a_4, a_6)$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const point<gf_element> &` I`.get_G () const`
: returns a point $G \in E(\mathbb{F}_q)$ of order r. The method `generate` has to be invoked before. Otherwise, 0 is returned.

Accessors for static variables.

`lidia_size_t` I`.get_default_lower_bound_bitlength_r () const`
: returns the default lower bound of bitlength of r.

`lidia_size_t` I`.get_default_upper_bound_k () const`
: returns the default upper bound of k.

`const bigint &` I`.get_default_lower_bound_extension_bitlength () const`
: returns the default lower bound of the bitlength of finite fields in which the discrete logarithm problem is considered intractable.

Accessors for non-static variables to set security level.

`lidia_size_t` I`.get_lower_bound_bitlength_r () const`
: returns the lower bound of bitlength of r.

`const bigint &` I`.get_upper_bound_k () const`
: returns the upper bound of k.

`const bigint & I.get_delta_field () const`
returns Δ_K.

`lidia_size_t I.get_h_field () const`
returns $h(\Delta_K)$.

`lidia_size_t I.get_lower_bound_h_field () const`
returns lower bound of $h(\Delta_K)$.

Accessors to get variables in the context of the requirements of the German Information Security Agency (GISA, BSI).

`bool I.get_according_to_BSI () const`
returns `true` if and only if the generated curve respects the requirements of GISA.

`lidia_size_t I.get_BSI_lower_bound_h_field () const`
returns the lower bound of the class number of the the maximal order containing the endomorphism ring of the curve.

`lidia_size_t I.get_BSI_lower_bound_extension_degree () const`
returns the lower bound of the order of q in $\mathbb{F}_r^\times$.

Accessors which are specific to `gec_complex_multiplication`.

`const bigint & I.get_delta () const`
returns Δ.

`lidia_size_t I.get_h () const`
returns $h(\Delta)$.

`unsigned int I.get_generation_mode () const`
returns the generation mode.

`long I.get_complex_precision () const`
returns the floating point precision to compute the class polynomial corresponding to `generation_mode`.

`const polynomial<bigint> & I.get_class_polynomial () const`
returns the class polynomial.

High-Level Methods and Functions

Let I be an instance of `gec_complex_multiplication`.

`void I.generate ()`
generates the field (if not already set) and the elliptic curve corresponding to the security level in use.

`void I.generate_oef ()`
generates the Optimal Extension Field (the degree has to be set by the user) and the elliptic curve of prime order over this field corresponding to the security level in use.

`void` *I*`.compute_class_polynomial_karatsuba ()`
computes the class polynomial corresponding to the chosen discriminant Δ and generation mode. The computation uses an efficient arithmetic basing on ideas of Karatsuba.

Input/Output

Input/Output of instances of `gec_complex_multiplication` is currently not possible.

See also

`gec_point_counting_mod_p`, `gec_point_counting_mod_2n`.

Notes

Author

Harald Baier

Bibliography

[AM93] A.O.L. Atkin and F. Morain. Elliptic curves and primality proving. *Mathematics of Computation*, 61:29–67, 1993.

[Apo90] T.M. Apostol. Modular Functions and Dirichlet Series in Number Theory. Springer-Verlag, 1990.

[Bai01a] H. Baier. Efficient Computation of Fourier Series and Singular Moduli with Application in Cryptography. Technical Report, Darmstadt University of Technology, 2001. Technical Report No. TI-7/01.

[Bai01b] H. Baier. Efficient Computation of Singular Moduli with Application in Cryptography. In *Fundamentals of Computing Theory, Proceedings of FCT 2001*, LNCS 2138, pages 71–82, Berlin, 2001. Springer-Verlag. 13th International Symposium, Riga, Latvia, August 22-24.

[Bai01c] H. Baier. Elliptic Curves of Prime Order over Optimal Extension Fields for Use in Cryptography. In *Progress in Cryptology - INDOCRYPT 2001*, LNCS 2247, pages 99–107, Berlin, 2001. Springer-Verlag. Second International Conference on Cryptology in India, Chennai, December 16-20.

[Bai01d] H. Baier. Elliptic Curves of Prime Order over Optimal Extension Fields for Use in Cryptography. Technical Report, Darmstadt University of Technology, 2001. Technical Report No. TI-11/01.

[BB00] H. Baier and J. Buchmann. Efficient Construction of Cryptographically Strong Elliptic Curves. In *Progress in Cryptology - INDOCRYPT2000*, LNCS 1977, pages 191–202, Berlin, 2000. Springer-Verlag. First International Conference on Cryptology in India, Calcutta, December 10-13.

[BP98] D. Bailey and C. Paar. Optimal Extension Fields for fast Arithmetic in Public-Key Algorithms. In *Advances in Cryptology - CRYPTO'98*, LNCS 1462, pages 472–485, Berlin, 1998. Springer-Verlag.

[BP01] D. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. *Journal of Cryptology*, 14:3:153–176, 2001.

[Bro93] I. Bronstein. Taschenbuch der Mathematik. Verlag Harri Deutsch, 1993.

[BS66] Z. Borevich and I. Shafarevich. Number Theory. Academic Press, 1966.

[BS96] E. Bach and J. Shallit. Algorithmic Number Theory, Volume I (Efficient Algorithms). MIT Press, 1996.

[BSS99] I. Blake, G. Seroussi, and N. Smart. Elliptic Curves in Cryptography. Cambridge University Press, 1999.

[Buc01] J. Buchmann. Algorithms for Binary Quadratic Forms. Springer-Verlag, 2001.

[Coh62] H. Cohn. Advanced Number Theory. Dover Publications, 1962.

[Coh95] H. Cohen. A Course in Computational Algebraic Number Theory. Springer-Verlag, 1995.

[Cox89] D. Cox. Primes of the form $x^2 + ny^2$. John Wiley & Sons, 1989.

[Eng99] A. Enge. Elliptic Curves and their Applications to Cryptography. Kluwer Academic Publishers, 1999.

[FGH01] M. Fouquet, P. Gaudry, and R. Harley. Finding Secure Curves with the Satoh-FGH Algorithm and an Eary-Abort Strategy. In *Proceedings of Eurocrypt 2001*, LNCS 2045, pages 14–29, Berlin, 2001. Springer-Verlag.

[FMR98] G. Frey, M.Müller, and H.-G. Rück. The Tate Pairing and the Discrete Logarithm Problem Applied to Elliptic Curve Cryptosystems. Technical Report, Institut für Experimentelle Mathematik, Universität Essen, 1998. Technical Report.

[FR94] G. Frey and H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves. *Mathematics of Computation*, 62:865–874, 1994.

[GHS01] P. Gaudry, F. Hess, and N.P. Smart. Constructive and descructive facets of Weil descent on elliptic curves. *Journal of Cryptology*, 2001. to appear.

[GIS01] Geeignete Kryptoalgorithmen, In Erfüllung der Anforderungen nach §17(1) SigG vom 16. Mai 2001 in Verbindung mit §17(2) SigV vom 22. Oktober 1997, July 2001. Bundesanzeiger Nr. 158 - Seite 18 562 vom 24. August 2001.

[GK86] S. Goldwasser and J. Kilian. Almost all primes can be quickly certified. In *Proceedings of 18th STOC*, pages 316–329, 1986.

[Gou97] F.Q. Gouvea. Non-ordinary primes: A story. *Exp. Math.*, 6/3:195–205, 1997.

[GS00] R. Gross and J.H. Smith. A Generalization of a Conjecture of Hardy and Littlewood to Algebraic Number Fields. *Rocky Mountain Journal of Mathematics*, 30:195–215, 2000.

[Hus87] D. Husemöller. Elliptic Curves. Springer-Verlag, 1987.

[JMS01] M. Jacobson, A. Menezes, and A. Stein. Solving Elliptic Curve Discrete Logarithm Problems Using Weil Descent. *Journal of the Ramanujan Mathematical Society*, 16:231–260, 2001.

[Kal86] B. KALISKI. A pseudorandom bit generator based on elliptic logarithms. In *Advances in Cryptology-CRYPTO'86*, LNCS 293, pages 84–103. Springer-Verlag, 1986.

[Kal88] B. KALISKI. *Elliptic curves and cryptography.* PhD thesis, M.I.T., 1988.

[Kan] M. KANEKO. Traces of singular moduli and the Fourier coefficients of the elliptic modular function $j(\tau)$. private communicated.

[Kar95] A. KARATSUBA. The complexity of computations. *Proc.Steklov Inst.Math.*, 211:169–183, 1995.

[Köh88] G. KÖHLER. Theta series on the Hecke groups $G(\sqrt{2})$ and $G(\sqrt{3})$. *Math. Z.*, 197/1:69–96, 1988.

[Köh02] G. KÖHLER. private communication, 2002.

[Knu98] J. KNUDSEN. JAVA Cryptography. O'Reilly, 1998.

[Kob87] N. KOBLITZ. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Kob93] N. KOBLITZ. Introduction to Elliptic Curves and Modular Forms. Springer-Verlag, 1993.

[Lan70] S. LANG. Algebraic Number Theory. Addison-Wesley, 1970.

[Lay94] G.-J. LAY. Konstruktion elliptischer Kurven mit gegebener Gruppenordnung über endlichen Primkörpern. Master's thesis, Universität des Saarlandes, 1994.

[Len87] H. LENSTRA. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[LiDIA] LiDIA. A library for computational number theory. Darmstadt University of Technology. URL: http://www.informatik.tu-darmstadt.de/TI/LiDIA/Welcome.html.

[Lip00] M. LIPPERT. Ein beweisbar sicherer Pseudozufallsbit-Generator auf der Basis des DL-Problems in elliptischen Kurven. Master's thesis, Darmstadt University of Technology, 2000.

[LZ94] G.-J. LAY AND H.G. ZIMMER. Constructing elliptic curves with given group order over large finite fields. In *Proceedings of ANTS I*, LNCS 877, pages 250–263, 1994.

[Mah76] K. MAHLER. On a class of non-linear functional equations connected with modular functions. *Journal of the Australian Mathematical Society*, 22, Series A:65–120, 1976.

[Men93] A. MENEZES. Elliptic Curve Public Key Cryptosystems. Kluwer Academic Publishers, 1993.

[Mil86] V. MILLER. Use of Elliptic Curves in Cryptography. In *Proceedings of CRYPTO '85*, LNCS 218, pages 417–426, Berlin, 1986. Springer-Verlag.

[Mül95] V. Müller. *Ein Algorithmus zur Bestimmung der Punktanzahl elliptischer Kurven über endlichen Körpern der Charakteristik größer drei.* PhD thesis, University of Saarbrücken, 1995.

[Mor88] F. Morain. Implementation of the Atkin-Goldwasser-Kilian primality testing algorithm. Technical Report, INRIA, 1988.

[MOV91] A. Menezes, T. Okamoto, and S. Vanstone. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 80–89, 1991.

[MP97] V. Müller and S. Paulus. On the Generation of Cryptographically Strong Elliptic Curves. Technical Report, Darmstadt University of Technology, 1997. Technical Report No. TI-25/97.

[Nie75] D. Niebur. A formula for Ramanujan's τ-function. *Ill. J. Math.*, 19:448–449, 1975.

[P1363] P1363 Standard Specifications for Public Key Cryptography. IEEE, Draft 13, 1999.

[Sat99] T. Satoh. The Canonical Lift of an Ordinary Elliptic Curve over a Finite Field and its Point Counting. http://www.rimath.saitama-u.ac.jp/lab.en/TkkzSatoh/, 1999.

[Sch53] B. Schoeneberg. Über den Zusammenhang der Eisensteinschen Reihen und Thetareihen mit der Diskriminante der elliptischen Funktionen. *Math. Ann.*, 126:177–184, 1953.

[Sem98] I. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Mathematics of Computation*, 67:353–356, 1998.

[Ser85] J.P. Serre. Sur la lacunarite des puissances de η. *Glasg. Math. J.*, 27:203–221, 1985.

[Sho95] V. Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20:363–397, 1995.

[Sil86] J.H. Silverman. The Arithmetic of Elliptic Curves. Springer-Verlag, 1986.

[Sma99] N.P. Smart. The Discrete Logarithm Problem on Elliptic Curves of Trace One. *Journal of Cryptology*, 12/3:193–196, 1999.

[Sma01] N.P. Smart. How Secure Are Elliptic Curves over Composite Extension Fields? In *Proceedings of Eurocrypt 2001*, LNCS 2045, pages 30–39, Berlin, 2001. Springer-Verlag.

[Spa92] A.-M. Spallek. Konstruktion einer elliptischen Kurve über einem endlichen Körper zu gegebener Punktegruppe. Master's thesis, University of Essen, 1992.

[SScK01] E. Savaş, A. Schmidt, and Ç. Koç. Generating Elliptic Curves of Prime Order. In *Proceedings of CHES '01*, LNCS 2, pages 145–161, Berlin, 2001. Springer-Verlag.

[SEC1] SEC1 Standards for Efficient Cryptography: Elliptic Curve Cryptography. Version 1.0, 2000.

[vOM99] P.C. van Oorschot and M.J.Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12/1:1–28, 1999.

[X9.62] X9.62 Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). ANSI, 1998.

[YZ97] N. Yui and D. Zagier. On the singular values of Weber modular functions. *Mathematics of Computation*, 66:1645–1662, 1997.

[Zag81] D. Zagier. Zetafunktionen und quadratische Körper. Springer-Verlag, Berlin, 1981.

Index

Curriculum Vitae (Academic Education)

15/03/71	Born in Fulda (Hessen), Germany
08/77 - 07/81	Basic education
08/81 - 06/90	Grammar school
06/90	Abitur, Freiherr-vom-Stein-Gymnasium Fulda
10/92 - 01/99	Studies in mathematics and physics at Julius-Maximilians-Universität Würzburg
01/99	First State Examination for Lectureship in Mathematics and Physics, Julius-Maximilians-Universität Würzburg
since 02/99	Research associate at Prof. Dr. J. Buchmann, Darmstadt University of Technology, Department of Computer Science